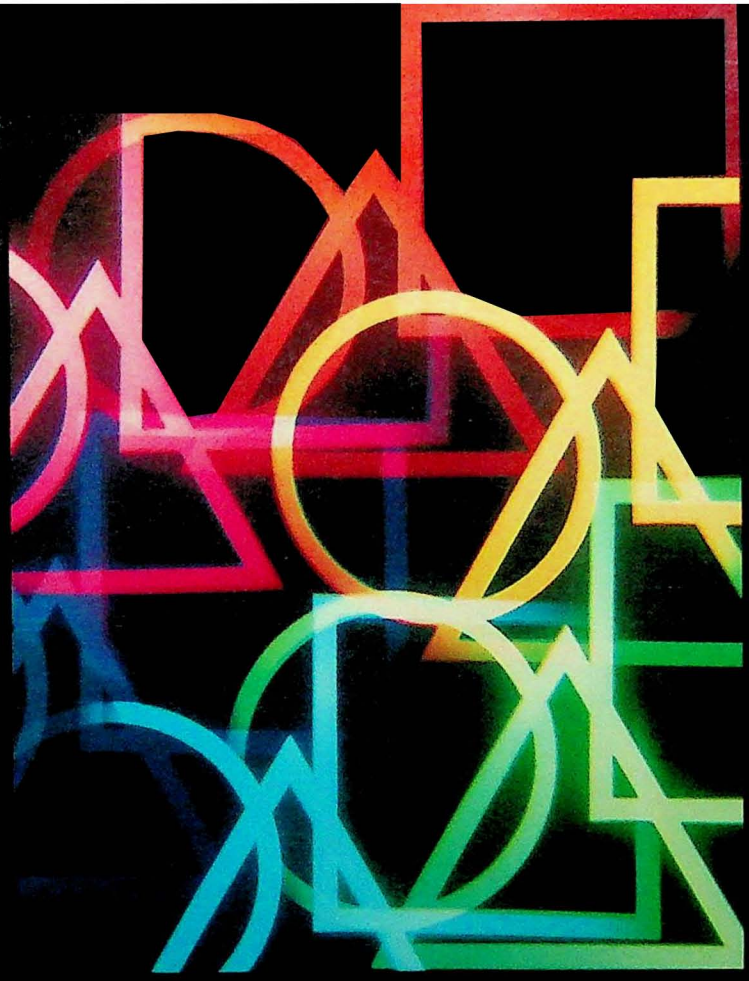


VAX Architecture Reference Manual

Second Edition

Richard A. Brunner

VAX Architecture Reference Manual

Contributing Authors

Dileep P. Bhandarkar
Peter F. Conklin
David N. Cutler
Thomas W. Eggers
Thomas N. Hastings
Richard I. Hustvedt
Judson S. Leonard
Timothy E. Leonard
Peter Lipman
Thomas Rarich
David P. Rodgers
Stephen Rothman
William D. Strecker
Theodore B. Taylor
Cheryl A. Wiecek

VAX Architecture Reference Manual

Second Edition

Edited by Richard A. Brunner

digital
Digital Press

Copyright © 1987, 1991 by Digital Equipment Corporation.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher.

9 8 7 6 5 4 3 2 1

Printed in the United States of America.

Order number EY-F576E-DP

PostScript is a trademark of Adobe Systems Inc.; TEX is a trademark of the American Mathematical Society; Unix is a trademark of AT&T Bell Laboratories; SYSTEM/360 and SYSTEM/370 are trademarks of International Business Machines Corporation; DEC, DECnet, the Digital logo, MASSBUS, MicroVAX, PDP, RSX, rtVAX, SBI, ULTRIX, UNIBUS, VAX, VAXstation, and VMS are trademarks of Digital Equipment Corporation.

Design: Sandra Calef
Production: Nancy Benjamin
Copy Editing: Nancy Wirtes
Proofreading: Alan Clark
Indexing: Marilyn Rowland
Typesetting: Paul C. Anagnostopoulos & Richard A. Wells, using the ZZTEX macro package for the TEX typesetting system (by Donald E. Knuth)
Reproduction Proof: INPRINT Publishing Corporation
Cover Photo: Earl J. Glass / Stock Boston
Printer: Hamilton Printing Company

Library of Congress Cataloging-in-Publication Data

VAX architecture reference manual / edited by Richard A. Brunner ; contributing authors, Dileep P. Bhandarkar ... [et al.]. — 2nd ed.
p. cm.
Includes index.
ISBN 1-55558-057-2
1. VAX computers. 2. Computer architecture. I. Brunner, Richard A. II. Bhandarkar, Dileep P.
QA76.8.V32V39 1991
004.1'45–dc20 90-49423
CIP

Contents

Foreword (First Edition) *ix*

Preface *xiii*

Introduction *1*
Terminology and Conventions *2*

1 ***Basic Architecture*** *4*
1.1 Addressing 4
1.2 Data Types 4
1.3 Processor State 19

2 ***Instruction Formats and Addressing Modes*** *25*
2.1 Opcode Formats 25
2.2 Operand Specifiers 25
2.3 Notation 27
2.4 General Mode Addressing Formats 27
2.5 Branch Mode Addressing Formats 39
2.6 Instruction Interpretation 40

3 ***Instructions*** *43*
3.1 Operand Specifier Notation 44
3.2 Operation Description Notation 45
3.3 Integer-Arithmetic and Logical Instructions 48
3.4 Address Instructions 67

3.5 *Variable-Length Bit-Field Instructions* 69
3.6 *Control Instructions* 74
3.7 *Procedure-Call Instructions* 91
3.8 *Miscellaneous Instructions* 99
3.9 *Queue Instructions* 110
3.10 *Floating-Point Instructions* 128
3.11 *Character-String Instructions* 153
3.12 *Cyclic Redundancy Check Instruction* 171
3.13 *Decimal-String Instructions* 175
3.14 *Edit Instruction* 198

4 **Memory Management** 213
4.1 *Virtual Address Space* 214
4.2 *Memory Management Control* 217
4.3 *Address Translation* 219
4.4 *Memory Protection* 223
4.5 *System-Space Address Translation* 226
4.6 *Process-Space Address Translation* 227
4.7 *Translation Buffer* 235
4.8 *Faults and Parameters* 238
4.9 *Privileged Services and Argument Validation* 239

5 **Interrupts and Exceptions** 244
5.1 *Processor Interrupt Priority Level and Interrupts* 245
5.2 *Exception Types* 246
5.3 *Interrupts versus Exceptions* 247
5.4 *Processor Status* 249
5.5 *Restartability* 249
5.6 *Interrupts* 250
5.7 *Exceptions* 254
5.8 *Serialization of Notification of Multiple Events* 267
5.9 *System Control Block* 268
5.10 *Stacks* 272
5.11 *Initiate Exception or Interrupt* 277
5.12 *Instructions Related to Exceptions and Interrupts* 283

6 **Process Structure** 287
6.1 *Process Context* 287
6.2 *Asynchronous System Traps* 290

6.3 *Process-Scheduling Interrupts* 292
6.4 *Process-Structure Instructions* 292

7 ***Memory and I/O*** 299
7.1 *Memory, Multiprocessing, and Interprocessor Communication* 299
7.2 *Instruction Stream Processing and Modification* 307
7.3 *Memory References* 311
7.4 *Cache* 315
7.5 *I/O Structure* 316

8 ***Privileged Registers*** 323
8.1 *Per-Process Registers and Context Switching* 323
8.2 *Stack-Pointer Images* 327
8.3 *MTPR and MFPR Instructions* 328

9 ***PDP-11 Compatibility Mode*** 336
9.1 *General Registers and Addressing Modes* 337
9.2 *The Stack* 340
9.3 *Processor Status Word* 341
9.4 *Instructions* 341
9.5 *Entering and Leaving Compatibility Mode* 373
9.6 *Native-Mode and Compatibility-Mode Registers* 373
9.7 *Compatibility-Mode Memory Management* 373
9.8 *Compatibility-Mode Exceptions and Interrupts* 374
9.9 *Tracing in Compatibility Mode* 375
9.10 *Unimplemented PDP-11 Traps* 377
9.11 *Compatibility-Mode I/O References* 377
9.12 *Processor Registers* 377
9.13 *Program Synchronization* 377

10 ***System Bootstrapping and Console*** 379
10.1 *System Bootstrapping* 379
10.2 *System Restart* 382
10.3 *System Powerfail and Recovery* 384
10.4 *Major System State Transitions* 385
10.5 *System Halted (Console I/O Mode)* 387
10.6 *System Running (Program I/O Mode)* 413

11 ***Architectural Subsetting*** *416*
11.1 Instruction-Set Options 416
11.2 Other Implementation Options 421
11.3 Instruction Emulation 424

12 ***VAX Vector Registers and Instructions*** *429*
12.1 Introduction 429
12.2 Registers 430
12.3 Vector Instruction Formats 438
12.4 Vector Instruction Classes 444
12.5 Vector Memory Access Instructions 445
12.6 Vector Integer Instructions 454
12.7 Vector Logical and Shift Instructions 459
12.8 Vector Floating-Point Instructions 461
12.9 Vector Edit Instructions 474
12.10 Miscellaneous Instructions 477
12.11 Assembler Notation 482

13 ***VAX Vector Execution Model and Exception Reporting*** *485*
13.1 Execution Model 485
13.2 Vector Processor Exceptions 498
13.3 Synchronization 509
13.4 Memory Management 520
13.5 Hardware Errors 521

A ***Opcode Assignments*** *523*
A.1 One-Byte Opcodes 523
A.2 Two-Byte Opcodes 527
A.3 Vector Opcodes 530

B ***Implementation Dependencies*** *531*
B.1 Implementation Options 532
B.2 Implementation 537

Index *547*

Foreword (First Edition)

By any practical measure, the VAX family of computers is one of the most successful series of computer systems ever developed. At the time of this writing, over 100,000 machines have been installed, ranging in size from the MicroVAX II to the VAX 8800—a number that even surpasses that for the pioneering IBM SYSTEM 360/370 series. The VAX design has been implemented from scratch over seven times in the past decade to capitalize on advances in technology as well as the changing needs of our customers. These different implementations have used a variety of technologies, organizational techniques, and configurations. And though work on the first machine began in 1975, we expect VAX computers to remain the backbone of Digital's product offerings for many years into the future.

To a considerable extent, the success of the VAX family is due to this book, the *VAX Architecture Reference Manual.* Not only does it describe a computer architecture that is outstanding in its own right, but it does so in a manner that is more unambiguous, precise, and complete than for any other computer architecture. With this document, diverse hardware groups throughout Digital have been able to create compatible machines using different technologies, at different periods of time, and in widely separated locations. The book has also served as the control document for approved design modifications and over the years we have both extended and provided for subsets of its content to improve performance and to pave the way for smaller, yet compatible, implementations.

Of course, the main reason for the success of the VAX family lies in the design itself. The VAX architecture is a computer architecture in the classic sense, a design for a hardware/software interface that is meant to remain consistent, from the point of view of a machine-language program, across machines of varying price, performance, and technology. Such a well-defined interface was first used by IBM in its SYSTEM 360/370 family of computers; it was described in 1964 in a seminal paper by G.A. Blauw and F.P. Brooks, Jr., two of its principal designers. At that time, six models had been announced, and all, according to the authors, were "logically identical.... Even though the allowable (I/O) channels or storage capacity may vary from model to model...the logical structure can be discussed without reference to specific models." [1]

This is precisely the goal we set for the VAX design. By defining an architecture that would apply to all members of the VAX family, hardware engineers would be free to build different hardware instaniations or implementations "up" to the specification, while application and system programmers could safely program "down" to it, confident that any program conforming to the specification would run on any present or future machine. That goal has, in fact, been achieved. Today, any program that conforms to the VAX architecture will run on any VAX with the necessary resources. And hardware engineers without getting involved in the details of software, can build new generations of VAXes, confident that the billions of dollars invested in existing VAX applications will not be jeopardized. As for the success of the VAX architecture itself, there are a number of reasons for its widespread acceptance and longevity.

One major reason is the enormous size of the VAX virtual address space. Lack of virtual address space has been the Achilles' heel of most computer architectures. Not long after we announced the first PDP-11 in 1969, we realized that customers were going to demand minicomputers with more than 64 kilobytes of memory, the maximum amount that can be addressed directly by a 16-bit address. We could see that relentless progress in memory chip densities was going to lead to a quadrupling of bits per chip

1. G.A. Blauw and F.P. Brooks, Jr., "The Structure of SYSTEM/360: Part 1—Outline of the Logical Structure," *IBM Systems Journal*, vol. 3, no. 2 (1964), pp. 119–135. This is the first published description of a commercial computer architecture with multiple implementations.

every three to four years. Increasing densities yield decreasing memory costs and computers at minicomputer prices would be able to have more than 64 Kbytes. So over the years we first extended the PDP-11's physical memory to 256 Kbytes and then to 2 megabytes. However, the *virtual address* of the PDP-11 remains at 64 Kbytes and the programmer often faces the tedious task of mapping, and then remapping, the PDP-11's small virtual address into a much larger physical memory.

The VAX acronym itself (which originally stood for Virtual Address eXtension) clearly indicated a major design goal of the project: to dramatically increase the address space of the popular PDP-11 computer architecture. The desire to build a machine with enough address space to satisfy customers for years to come led to the decision to create a new 32-bit architecture. With 32-bit addresses, 4 billion bytes of address space were available. The first VAX-11/780 machines shipped in early 1978 with one quarter of a megabyte of physical memory built from 4K-bit memory chips. By contrast the VAX 8650, one of our more recent large computers, can be configured with 68 megabytes of physical memory, built this time for 256K-bit memory chips. Each time the physical memory of a machine quadruples, an additional 2 bits of address are required to reference it. To address 64 million bytes on a VAX 8650, 26 bits are needed. If the density of memory chips continues to quadruple every three to four years, then the 32-bit address of the VAX architecture will be adequate for at least another decade without requiring programmers to map virtual memory onto a larger physical memory.

In addition to its expanded address space, the VAX architecture built upon the elegant instruction set and addressing modes of the PDP-11. Additional data formats and corresponding instructions were added to support the needs of compiler writers, as well as scientific and commercial application programmers. A standard calling interface was designed to allow modules written in different langauges to call one another. And finally, most importantly, the architecture was carefully designed to support the needs of a modern virtual-memory operating system.[2]

2. Lawrence Kenah and Simon F. Bate, *VAX/VMS Internals and Data Structures* (Digital Press, 1984) provides a thorough discussion of the algorithms and data structures of the VAX/VMS operating system, including their interactions.

The result of this design work has been gratifying to all of us who have contributed. Customers can choose among three operating systems: VMS, Digital's operating system designed to take full advantage of the VAX architecture; ULTRIX, an implementation of the industry-standard UNIX operating system; and ELN, a system designed to support the development of dedicated, real-time applications. Programmers can choose from a large family of industry-standard, compatible languages, all of which make use of the calling standard and can access the supporting library routines and system services. The architecture has also allowed development of DECnet, Digital's network architecture and an incredibly rich set of Digital, third party, and customer applications.

An architectural specification can make for dry reading. Nevertheless, this book should be of real interest to at least three audiences. For the serious computer engineer who aspires to design a machine as good as (or better than) a VAX, the *VAX Architecture Reference Manual* is an outstanding example of a successful computer architecture and how it should be documented. For the serious application or systems programmer of VAX computers, this is also the book of "last resort," providing the most precise, authoritative, and complete description of the machine language interface with which he or she will work. Finally, for the serious students of either computer science or engineering, the *VAX Architecture Reference Manual* is an excellent supplementary reference, to be consulted as a case study in design or for additional detail regarding computer organization or assembly language programming.

Computer design continues to be a dynamic field; I expect we will see more rather than less change and innovation in the decades ahead. No matter how computers evolve, however, it is clear that the VAX architecture is a major contribution to progress in the field. It will be as important to study and understand a generation from now as it is today.

Samuel H. Fuller, Vice President, Research & Architecture
Digital Equipment Corporation
June 1986

Preface

VAX is a family of upward-compatible computer systems. It is a natural outgrowth of and is strongly compatible with the PDP-11 family.

This book is the definition of the VAX architecture. It provides a complete description of the VAX central-processor hardware as seen by machine-language programs, and applies to all software written for VAX processors, all VAX central-processor hardware, and all "closely coupled" VAX hardware peripherals. The definition of the central processor describes exactly what its instructions do, how it handles data, what its control and status information means, and what programming techniques and procedures must be employed to use it effectively. The programming is given in machine language, in that it uses only the basic instruction mnemonics and symbolic addressing defined by the assembler. Moreover, the book is completely self-contained—no prior knowledge of the assembler is required.

Even as the last edition of this book was being produced, Digital was already extending the VAX architecture in a number of key areas.

- Integrated vector processing. Sixteen vector registers and 63 new vector instructions were added as an integral part of the VAX architecture. Both the VAX 9000 and the VAX 6000 model 400 optionally support it.
- 34-bit extended physical address mode. The architecture now allows new VAX processors to address up to 16 gigabytes of

physical memory. This feature allows VAX 9000 systems with the proper memory technology to have gigabytes of physical memory.

- With the advent of numerous VAX multiprocessor systems, the rules for sharing data among processors have been refined and elucidated. The underlying concepts of atomicity, read-write-ordering, and corruption ("word-tearing") have been established.
- The subtle and not-so-subtle means of instruction stream modification (self-modifying code) have been further identified. Methods for ensuring correct operation in all cases of instruction stream modification have been established.

This edition elaborates not only on the preceding features but also contains a number of minor changes and corrections that are typical for an architecture that is as active and as long-lived as the VAX architecture.

Acknowledgments

Many people contribute to the making of a book; I thank the many who made this book possible. I wish that I could acknowledge each of them here for the help I have received. But, the limitations of space allow me to acknowledge only a few.

No book can be written without a subject; I owe mine to the inventors and refiners of the VAX architecture, many of whose names appear as contributing editors to this book. They have my appreciation for a job well done. I especially thank Dileep Bhandarkar and Bob Supnik for supplying me with some wonderful source material on VAX processor implementations for this edition.

I thank Mike Meehan and Chase Duffy at Digital Press for getting this book started. And I compliment Nancy Benjamin, Sandra Calef, and Paul Anagnostopoulos for the fine way in which they produced the book. I especially thank Paul, whose prowess with TEX and mastery of the book's subject matter gave me great confidence in the quality and the accuracy of this edition.

While making this book, I had much support and encouragement. I thank Audrey Reith for her support of this effort through the many months that I took to finish the book while in her employ.

I also thank my parents Alferd and Carolyn Brunner, whose continual, unceasing encouragement and support from the beginning have made all of this possible.

Most of all, I thank my wife, Chris, who put up with the long hours, the complaining, and sleepless nights. Her love gave me the renewed strength each day to finish this book. It is to her that I dedicate this book.

Richard A. Brunner
August 1990

Introduction

The VAX architecture represents a significant extension of the PDP-11 family architecture. It shares byte addressing with the PDP-11, similar I/O and interrupt structures, and identical data formats. Although the instruction set is not strictly compatible with the PDP-11, it is related and can be mastered easily by a PDP-11 programmer. Likewise, the similarity allows straightforward manual conversion of existing PDP-11 programs to the VAX system. Existing user-mode PDP-11 programs that do not need the extended features of VAX can run unchanged in the PDP-11 compatibility mode provided in the VAX architecture.

As compared to the PDP-11, VAX offers a greatly extended virtual address space, additional instructions and data types, and new addressing modes. VAX architecture also provides a sophisticated memory management and protection mechanism, and hardware-assisted process scheduling and synchronization.

A number of specific goals are achieved in the VAX design:

- VAX architecture has maximal compatibility with the PDP-11 consistent with a significant extension of the virtual address space and a significant functional enhancement.
- High bit efficiency is achieved by a wide range of data types and new addressing modes.
- The systematic, elegant instruction set with orthogonality of operators, data types, and addressing modes can be exploited easily, particularly by high-level language processors.
- The VAX system is extensible. The instruction set is designed

so that new data types and operators can be included efficiently in a manner consistent with the currently defined operators and data types. (A recent example of this extensibility is the addition of integrated vector processing.)

- The architecture is suitable in terms of price and performance over a wide range of computer system implementations sold by Digital Equipment Corporation.

Terminology and Conventions

Certain terminology and conventions used in this book are described in the following subsections.

Numbering

All numbers unless otherwise indicated are decimal. Where there is ambiguity, the radix is explicitly stated, as in 48 (hex) or 1001 (binary).

Scalar versus Vector

With the addition of vector processing, the VAX architecture can be classified into two parts: a vector part, which includes the instructions, registers, and execution model for vector processing, and a scalar part, which includes the remainder of the architecture. Where confusion may be possible, this book uses the term *scalar* to describe objects belonging to the scalar part of the architecture, as in "scalar instructions" and "scalar processor." Similarly, the term *vector* is used to describe objects belonging to the vector part of the architecture, as in "vector registers" and "vector instructions." With the exception of Chapters 12 and 13, instructions, exceptions, registers, and other objects described in the rest of this book refer to the scalar part of the architecture unless otherwise stated.

UNPREDICTABLE and UNDEFINED

Results specified as UNPREDICTABLE may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. Software can never depend on results specified as UNPREDICTABLE. Operations specified as UNDEFINED may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. The operation may vary in effect from nothing to stopping system operation. UNDEFINED operations must not

cause the processor to hang (reach an unhalted state, from which there is no transition to a normal state in which the processor executes instructions). Note the distinction between result and operation: non-privileged software cannot invoke UNDEFINED operations. When the operation of the VAX scalar processor becomes UNDEFINED, so does the operation of its associated vector processor. The converse is not true; when the operation of the vector processor becomes UNDEFINED, the operation of the scalar processor need not become UNDEFINED.

Ranges and Extents

Ranges are specified in English and are inclusive. For example, a range of integers 0 through 4 includes the integers 0, 1, 2, 3, and 4. Extents are specified by a pair of numbers separated by a colon and are inclusive. For example, bits ⟨7:3⟩ specifies an extent of bits including bits 7, 6, 5, 4, and 3.

MBZ

Fields specified as MBZ (Must Be Zero) should never be filled by software with a non-zero value. If the processor encounters a non-zero value in a field specified as MBZ, a reserved operand fault or abort occurs if that field is accessible to non-privileged software. (See Chapter 5 for a description of faults and aborts.) MBZ fields that are accessible only to privileged software (kernel mode) may not be checked for non-zero value by some or all VAX implementations. Non-zero values in MBZ fields accessible only to privileged software may produce UNDEFINED operation.

SBZ

Fields specified as SBZ (Should Be Zero) can produce UNPREDICTABLE results when filled by software with non-zero values. Unassigned values of fields are reserved for future use. In many cases, some values are indicated as reserved for the customer, that is, the equipment owner. Only these values should be used for non-standard applications. The values indicated as reserved for DIGITAL and all MBZ fields are to be used only to extend the standard architecture in the future.

Figure Conventions

Figures depicting registers or memory follow the convention that increasing addresses run right to left and top to bottom.

1

Basic Architecture

1.1 Addressing

The basic addressable unit in the VAX architecture is the 8-bit byte. Virtual addresses are 32 bits long: hence, the virtual address space is 2^{32} (approximately 4.3 billion) bytes. Virtual addresses as seen by the program are translated into physical memory addresses by the memory-management mechanism described in Chapter 4.

1.2 Data Types

Following are descriptions of the VAX architecture data types.

1.2.1 Byte

A byte is 8 contiguous bits starting on an addressable byte boundary. The bits are numbered from the right ⟨0⟩ through ⟨7⟩, as shown in Figure 1.1. A byte is specified by its address A. When interpreted arithmetically, a byte is a two's complement integer with bits of increasing significance from ⟨0⟩ through ⟨6⟩ and bit ⟨7⟩, the sign bit. The value of the integer is in the range −128 through 127. For the purposes of addition, subtraction, and comparison, VAX instructions also provide direct support for the interpretation of a byte as an unsigned integer with bits of increasing significance from ⟨0⟩ through ⟨7⟩. The value of the unsigned integer is in the range 0 through 255.

1.2.2 Word

A word is 2 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right ⟨0⟩ through ⟨15⟩. See

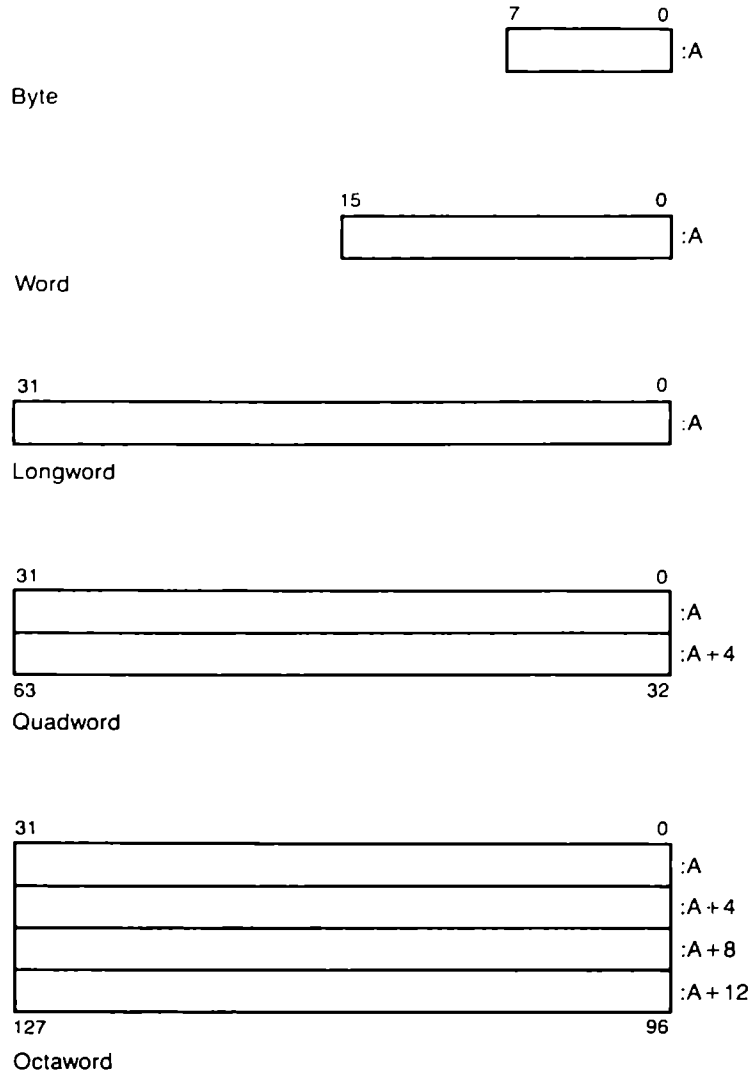


Figure 1.1
Data Types

Figure 1.1. A word is specified by its address A, the address of the byte containing bit ⟨0⟩. When interpreted arithmetically, a word is a two's complement integer with bits of increasing significance from ⟨0⟩ through ⟨14⟩ and bit ⟨15⟩, the sign bit. The value of the integer is in the range –32,768 through 32,767. For the purposes of addition, subtraction, and comparison, VAX instructions also provide direct support for the interpretation of a word as an unsigned integer with bits of increasing significance from ⟨0⟩ through ⟨15⟩. The value of the unsigned integer is in the range 0 through 65,535.

1.2.3 Longword

A longword is 4 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right ⟨0⟩ through ⟨31⟩, as shown in Figure 1.1. A longword is specified by its address A, the address of the byte containing bit ⟨0⟩. When interpreted arithmetically, a longword is a two's complement integer with bits of increasing significance from ⟨0⟩ through ⟨30⟩ and bit ⟨31⟩, the sign bit. The value of the integer is in the range –2,147,483,648 through 2,147,483,647. For the purposes of addition, subtraction, and comparison, VAX instructions also provide direct support for the interpretation of a longword as an unsigned integer with bits of increasing significance from ⟨0⟩ through ⟨31⟩. The value of the unsigned integer is in the range 0 through 4,294,967,295.

1.2.4 Quadword

A quadword is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right ⟨0⟩ through ⟨63⟩, as shown in Figure 1.1. A quadword is specified by its address A, the address of the byte containing bit ⟨0⟩. When interpreted arithmetically, a quadword is a two's complement integer with bits of increasing significance from ⟨0⟩ through ⟨62⟩ and bit ⟨63⟩, the sign bit. The value of the integer is in the range -2^{63} to $2^{63}-1$. Only a subset of the full complement of operators is provided for quadword.

1.2.5 Octaword

(The octaword data type need not be implemented by a processor. For more detail about implementation options, see Chapter 11.) An octaword is 16 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right ⟨0⟩ through ⟨127⟩, as shown in Figure 1.1. An octaword is specified by its address A, the address of the byte containing bit ⟨0⟩. When interpreted arithmetically, an octaword is a two's complement integer with bits of increasing significance from ⟨0⟩ through ⟨126⟩ and bit ⟨127⟩, the sign bit. The value of the integer is in the range -2^{127} to $2^{127}-1$. Only a subset of the full complement of operators is provided for octaword.

1.2.6 *F_floating*

An F_floating datum is 4 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right ⟨0⟩ through ⟨31⟩, as shown in Figure 1.2. An F_floating datum is specified by its address A, the address of the byte containing bit ⟨0⟩. The form of an F_floating datum is sign magnitude with bit ⟨15⟩, the sign bit; bits ⟨14:7⟩, an excess 128 binary exponent; and bits ⟨6:0⟩ and ⟨31:16⟩, a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go from ⟨16⟩ through ⟨31⟩ and ⟨0⟩ through ⟨6⟩. The 8-bit exponent field encodes the values 0 through 255. An exponent value of 0 together with a sign bit of 0 is taken to indicate that the F_floating datum has a value of 0. Exponent values of 1 through 255 indicate true binary exponents of −127 through +127. An exponent value of 0 together with a sign bit of 1 is taken as reserved. Floating-point instructions processing a reserved operand take a reserved operand fault. (See Chapter 5 for a description of faults.) The value of an F_floating datum is in the approximate range 0.29×10^{-38} through 1.7×10^{38}. The precision of an F_floating datum is approximately one part in 2^{23}, typically 7 decimal digits.

1.2.7 *D_floating*

A D_floating datum is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right ⟨0⟩ through ⟨63⟩, as shown in Figure 1.2. A D_floating datum is specified by its address A, the address of the byte containing bit ⟨0⟩. The form of a D_floating datum is identical to an F_floating datum except for an additional 32 low-significance fraction bits. Within the fraction, bits of increasing significance are from ⟨48⟩ through ⟨63⟩, ⟨32⟩ through ⟨47⟩, ⟨16⟩ through ⟨31⟩, and ⟨0⟩ through ⟨6⟩. The exponent conventions and approximate range of values is the same for D_floating as for F_floating. The precision of a D_floating datum is approximately one part in 2^{55}, typically 16 decimal digits.

1.2.8 *G_floating*

A G_floating datum is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right ⟨0⟩ through ⟨63⟩, as shown in Figure 1.2. A G_floating datum is specified by its address A, the address of the byte containing bit ⟨0⟩. The form of a G_floating datum is sign magnitude with bit ⟨15⟩, the sign bit; bits ⟨14:4⟩, an excess 1024 binary exponent; and bits ⟨3:0⟩ and ⟨63:16⟩, a normalized 53-bit fraction with the redundant most significant

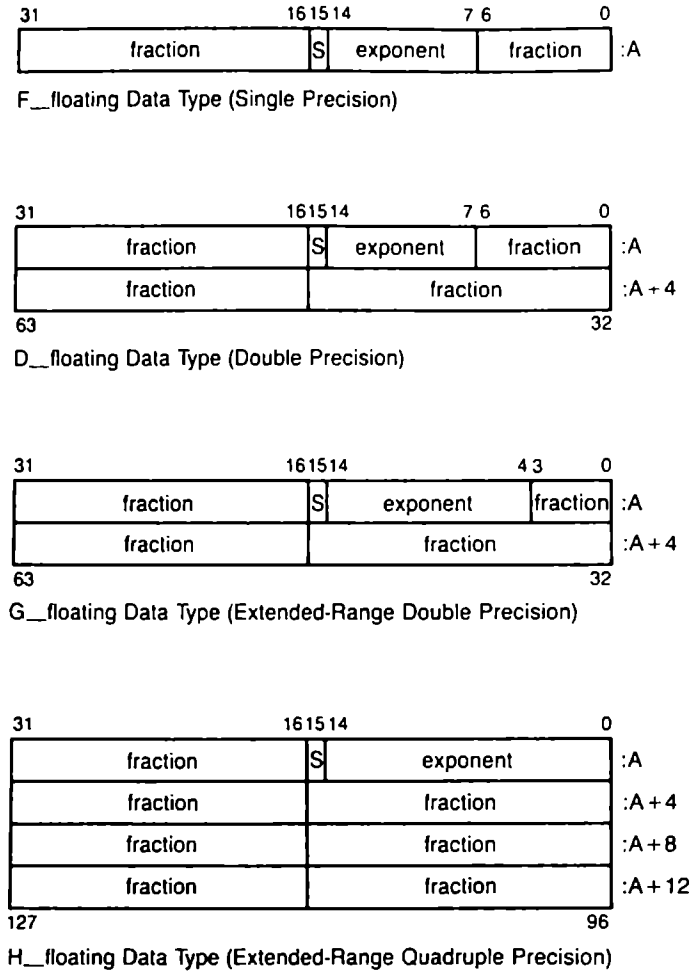


Figure 1.2
Floating-Point Data Types

fraction bit not represented. Within the fraction, bits of increasing significance are from ⟨48⟩ through ⟨63⟩, ⟨32⟩ through ⟨47⟩, ⟨16⟩ through ⟨31⟩, and ⟨0⟩ through ⟨3⟩. The 11-bit exponent field encodes the values 0 through 2047. An exponent value of 0 together with a sign bit of 0 is taken to indicate that the G_floating datum has a value of 0. Exponent values of 1 through 2047 indicate true binary exponents of −1023 through +1023. An exponent value of 0 together with a sign bit of 1 is taken as reserved. Floating-point instructions processing a reserved operand take a reserved operand fault. (See Chapter 5 for a description of faults.) The value of a G_floating datum is in the approximate range 0.56×10^{-308} through

0.9×10^{308}. The precision of a G_floating datum is approximately one part in 2^{52}, typically 15 decimal digits.

1.2.9 *H_floating*

(The H_floating data type need not be implemented by a processor. For more detail about implementation options, see Chapter 11.) An H_floating datum is 16 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right ⟨0⟩ through ⟨127⟩, as shown in Figure 1.2. An H_floating datum is specified by its address A, which is the address of the byte containing bit ⟨0⟩. The form of an H_floating datum is sign magnitude with bit ⟨15⟩, the sign bit; bits ⟨14:0⟩, an excess 16384 binary exponent; and bits ⟨127:16⟩, a normalized 113-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance are from ⟨112⟩ through ⟨127⟩, ⟨96⟩ through ⟨111⟩, ⟨80⟩ through ⟨95⟩, ⟨64⟩ through ⟨79⟩, ⟨48⟩ through ⟨63⟩, ⟨32⟩ through ⟨47⟩, and ⟨16⟩ through ⟨31⟩. The 15-bit exponent field encodes the values 0 through 32767. An exponent value of 0 together with a sign bit of 0 is taken to indicate that the H_floating datum has a value of 0. Exponent values of 1 through 32767 indicate true binary exponents of −16383 through +16383. An exponent value of 0 together with a sign bit of 1 is taken as reserved. Floating-point instructions processing a reserved operand take a reserved operand fault. (See Chapter 5 for a description of faults.) The value of an H_floating datum is in the approximate range 0.84×10^{-4932} through 0.59×10^{4932}. The precision of an H_floating datum is approximately one part in 2^{112}, typically 33 decimal digits.

1.2.10 *Variable-Length Bit Field*

A variable-length bit field is 0 to 32 contiguous bits located arbitrarily with respect to byte boundaries. A variable-length bit field is specified by three attributes as shown in Figure 1.3: the address A of a byte; a bit position P, which is the starting location of the field with respect to bit ⟨0⟩ of the byte at A; and a size S of the field.

For bit fields in memory, the position is in the range -2^{31} through $2^{31} - 1$ and is conveniently viewed as a signed 29-bit byte offset and a 3-bit bit-within-byte field, as shown in Figure 1.3. The sign-extended 29-bit byte offset is added to the address A, and the resulting address specifies the byte in which the field begins.

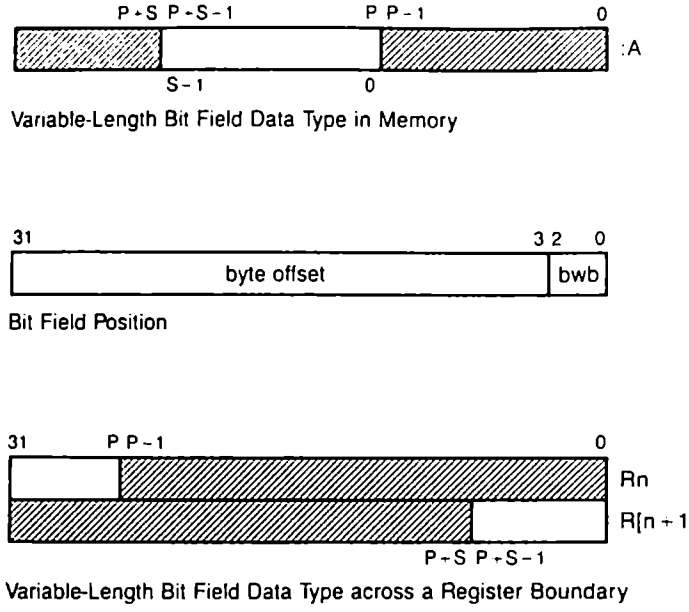


Figure 1.3
Variable-Length Bit Fields

The 3-bit bit-within-byte field encodes the starting position (0 through 7) of the field within that byte. The VAX field instructions provide direct support for the interpretation of a field as a signed or unsigned integer. When interpreted as a signed integer, a field contains a two's complement integer with bits of increasing significance from 0 through S–2; bit S–1 is the sign bit. When interpreted as an unsigned integer, bits of increasing significance are from 0 to S–1. A field of size 0 has a value of 0.

A variable-length bit field may be contained in 1 to 5 bytes. Instructions that read or write fields in memory reference only the minimum number of aligned longwords necessary to contain the field. (For more detail about how fields are referenced in memory, see section 7.3.)

For bit fields in registers, the position is in the range 0 through 31. The position operand specifies the starting position (0 through 31) of the field in the register. A variable-length bit field may be contained in two registers if the sum of position and size exceeds 32, as shown in Figure 1.3.

See the descriptions of the bit-field instructions in section 3.5 for further details on the specification of variable-length bit fields.

1.2.11 *Absolute Queues*

A queue is a circular, doubly linked list. A queue entry is specified by its address. Each queue entry is linked to the next with a pair of longwords. A queue is classified by the type of link it uses. Absolute queues use absolute addresses as links. The first (lowest addressed) longword is the forward link; it specifies the address of the succeeding queue entry. The second (highest addressed) longword is the backward link; it specifies the address of the preceding queue entry.

A queue is specified by a queue header, which is identical to a pair of queue linkage longwords. The forward link of the header is the address of the entry termed the head of the queue. The backward link of the header is the address of the entry termed the tail of the queue. The forward link of the tail points to the header.

An empty queue is specified by its header at address H, as shown in Figure 1.4. If an entry at address B is inserted into an empty queue (at either the head or tail), the second queue shown in Figure 1.4 results. The last three queues in Figure 1.4 illustrate the results of subsequent insertion of an entry at address A at the head, insertion of an entry at address C at the tail, and removal of the entry at address B.

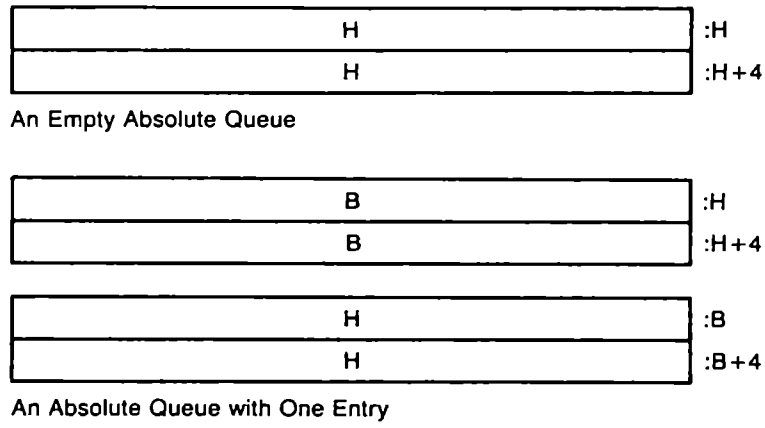
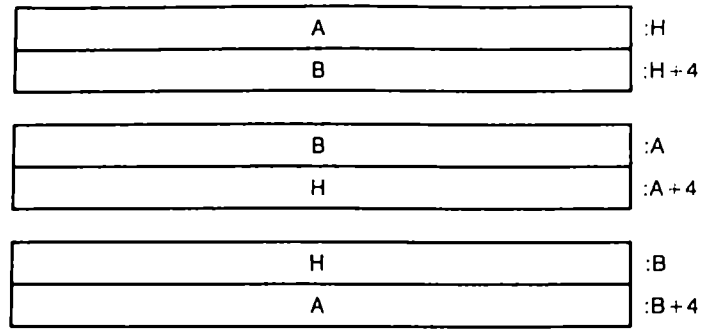
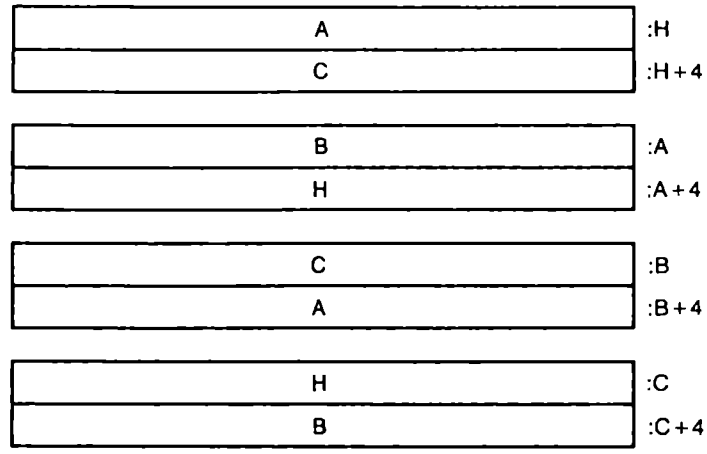


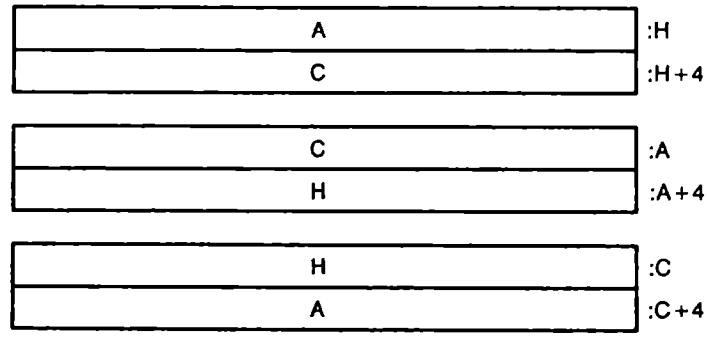
Figure 1.4
Absolute Queues (Part 1)



An Absolute Queue with Two Entries



An Absolute Queue with Three Entries



An Absolute Queue with Three Entries After Removing the Second Entry

Figure 1.4
Absolute Queues (Part 2)

1.2.12 *Self-Relative Queues*

Self-relative queues use displacements from queue entries as links. Queue entries are linked by a pair of longwords. The first longword (lowest addressed) is the forward link; it is a displacement of the succeeding queue entry from the present entry. The second longword (highest addressed) is the backward link; it is the displacement of the preceding queue entry from the present entry. A queue is specified by a queue header, which also consists of two longword links.

An empty queue is specified by its header at address H. Since the queue is empty, the self-relative links are zero, as shown in Figure 1.5.

The remainder of Figure 1.5 illustrates the results of subsequent insertion of an entry at address B at the head, insertion of an entry at address A at the tail, and insertion of an entry at address C at the tail.

1.2.13 *Character String*

A character string is a contiguous sequence of bytes in memory. A character string is specified by attributes: the address A of the first byte of the string and the length L of the string in bytes. The address of a string specifies the first character of a string. See Figure 1.6.

The length L of a string is in the range 0 through 65,535.

1.2.14 *Trailing Numeric String*

(The trailing-numeric-string data type need not be implemented by a processor. For more detail about implementation options, see Chapter 11.) A trailing numeric string is a contiguous sequence of bytes in memory. The string is specified by attributes: the address A of the first byte (most significant digit) of the string and the length L of the string in bytes.

All bytes of a trailing numeric string, except the least significant digit byte, must contain an ASCII decimal digit character (0–9). The highest addressed byte of a trailing numeric string represents an encoding of both the least significant digit and the sign of the numeric string.

0	:H
0	:H + 4

An Empty Self-Relative Queue

B - H	:H
B - H	:H + 4

H - B	:B
H - B	:B + 4

A Self-Relative Queue with One Entry

A - H	:H
B - H	:H + 4

B - A	:A
H - A	:A + 4

H - B	:B
A - B	:B + 4

A Self-Relative Queue with Two Entries

A - H	:H
C - H	:H + 4

B - A	:A
H - A	:A + 4

C - B	:B
A - B	:B + 4

H - C	:C
B - C	:C + 4

A Self-Relative Queue with Three Entries

Figure 1.5
Self-Relative Queues

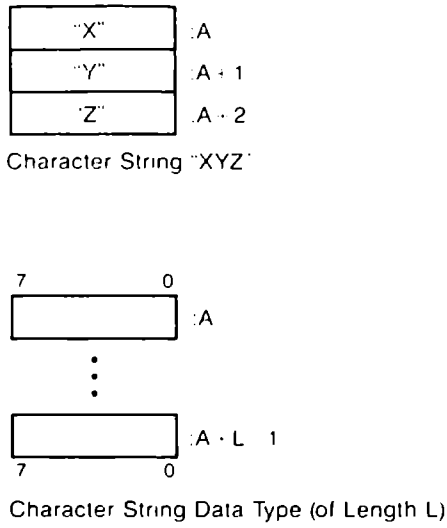


Figure 1.6
Character-String Data Type

The VAX numeric string instructions are able to support any encoding through the use of tables. There are, however, three preferred encodings used by DIGITAL software: (1) unsigned numeric, in which there is no sign and the least significant digit contains an ASCII decimal digit character, (2) zoned numeric, and (3) overpunched numeric. Because the overpunched format has been used by many compiler manufacturers over many years, and because various card encodings are used, several variations in overpunched format have evolved. Digital software accepts these alternate forms on input, but generates the normal form as output. The encoding of sign and digits in trailing numeric strings is shown in Table 1.1.

The length L of a trailing numeric string must be in the range 0 to 31 digits. The value of a zero-length string is zero.

The address A of the string specifies the byte of the string containing the most significant digit. Digits of decreasing significance are assigned to increasing addresses. Figure 1.7 illustrates the representation of trailing numeric strings in zoned and overpunched formats.

7 4	3 0	
3	1	:A
3	2	:A+1
3	3	:A+2

7 4	3 0	
3	1	:A
7	2	:A+1

Representation of "+123" and "−12" in Zoned Format

7 4	3 0	
3	1	:A
3	2	:A+1
3	3	:A+2

7 4	3 0	
3	1	:A
4	B	A+1

Representation of "+123" and "−12" in Overpunch Format

7 4	3 0	
2	B	:A
3	1	:A+1
3	2	:A+2
3	3	:A+3

7 4	3 0	
2	D	:A
3	1	:A+1
3	2	:A+2

Representation of "+123" and "−12" in Leading Separate Numeric String

7 4	3 0	
1	2	:A
3	12	:A+1

7 4	3 0	
0	1	:A
2	13	:A+1

Representation of "123" and "−12" in Packed Decimal String

Figure 1.7
Decimal Strings

1.2.15 *Leading-Separate Numeric String*

(The leading-separate-numeric-string data type need not be implemented by a processor. For more detail about implementation options, see Chapter 11.) A leading-separate numeric string is a contiguous sequence of bytes in memory. A leading-separate numeric string is specified by attributes: the address A of the first byte (containing the sign character) and a length L, which is the length of the string in digits and not the length of the string in bytes. The number of bytes in a leading-separate numeric string is L+1.

The sign of a leading-separate numeric string is stored in a separate byte. Each subsequent byte contains an ASCII digit character. The signs and digits of leading-separate numeric strings are shown in Table 1.1.

Table 1.1
Sign and Digits in Decimal Strings (Part 1)

	Zoned Trailing Numeric		Overpunched Trailing Numeric		Leading Separate Numeric		Packed Decimal
	Hex	ASCII	Hex	ASCII	Hex	ASCII	Hex
Sign							
positive					2B	+	C
positive[a]					20	⟨blank⟩	A E F
negative					2D	–	D
negative[a]							B
Digit							
0	30	0	30	0	30	0	0
1	31	1	31	1	31	1	1
2	32	2	32	2	32	2	2
3	33	3	33	3	33	3	3
4	34	4	34	4	34	4	4
5	35	5	35	5	35	5	5
6	36	6	36	6	36	6	6
7	37	7	37	7	37	7	7
8	38	8	38	8	38	8	8
9	39	9	39	9	39	9	9

[a]These alternative representations of the sign are permitted. VAX instructions always produce the preferred representation, which is shown first.

The length L of a leading-separate numeric string must be in the range 0 to 31 digits. The value of a zero-length string is zero.

The address A of the string specifies the byte of the string containing the sign. Digits of decreasing significance are assigned to bytes of increasing addresses. Figure 1.7 illustrates the representation of leading-separate numeric strings.

1.2.16 *Packed Decimal String*

(The packed-decimal-string data type need not be implemented by a processor. For more detail about implementation options, see Chapter 11.) A packed decimal string is a contiguous sequence of bytes in memory. A packed decimal string is specified by two attributes: the address A of the first byte of the string and a length L, which is the number of digits in the string and not the length

Table 1.1
Sign and Digits in Decimal Strings (Part 2)

	Zoned Trailing Numeric		Overpunched Trailing Numeric		
	Hex	ASCII	Hex	ASCII	Alt. ASCII
Combined sign and digit					
+0	30	0	7B	{	0 [?
+1	31	1	41	A	1
+2	32	2	42	B	2
+3	33	3	43	C	3
+4	34	4	44	D	4
+5	35	5	45	E	5
+6	36	6	46	F	6
+7	37	7	47	G	7
+8	38	8	48	H	8
+9	39	9	49	I	9
-0	70	p	7D	}	] ! :
-1	71	q	4A	J	
-2	72	r	4B	K	
-3	73	s	4C	L	
-4	74	t	4D	M	
-5	75	u	4E	N	
-6	76	v	4F	O	
-7	77	w	50	P	
-8	78	x	51	Q	
-9	79	y	52	R	

of the string in bytes. The bytes of a packed decimal string are divided into two 4-bit fields that must contain decimal digits, with the exception of the low nibble (bits (3:0)) of the last (highest addressed) byte that must contain a sign.

The preferred sign representation is 0C (hex) for positive and 0D (hex) for negative, as shown in Table 1.1.

The length L is the number of digits in the packed decimal string (not counting the sign) and must be in the range 0 through 31. When the number of digits is odd, the digits and the sign fit in L/2 (integer part only) + 1 bytes. When the number of digits is

even, it is required that an extra 0 digit appear in the high nibble (bits ⟨7:4⟩) of the first byte of the string. Again, the length in bytes of the string is L/2 + 1.

The address A of the string specifies the byte of the string containing the most significant digit in its high nibble. Digits of decreasing significance are assigned to increasing byte addresses and from high nibble to low nibble within a byte. Figure 1.7 illustrates the representation of packed decimal strings.

1.3 Processor State

The processor state consists of that portion of a process's state that, while the process is executing, is stored in processor registers rather than memory.

- Sixteen 32-bit general-purpose registers denoted Rn or R[n], where n is in the range 0 through 15
- A 32-bit processor status longword (PSL)
- Privileged internal processor registers (IPR)

1.3.1 General-Purpose Registers

The general-purpose registers are used as temporary storage, accumulators, index registers, and base registers. A register that contains an address is termed a base register. A register that contains an address offset is termed an index register. The bits of a register are numbered from the right ⟨0⟩ through ⟨31⟩, as shown in Figure 1.8.

Certain registers have special uses:

- R15 is the program counter (PC). PC contains the address of the next instruction byte of the program.
- R14 is the stack pointer (SP). SP contains the address of the top of the processor-defined stack.
- R13 is the current frame pointer (FP). The VAX procedure call convention builds a data structure on the stack called a stack frame. FP contains the address of the base of this data structure. (For more information about the VAX procedure calling convention, see the *VAX/VMS Run Time Library Reference Manual.*)

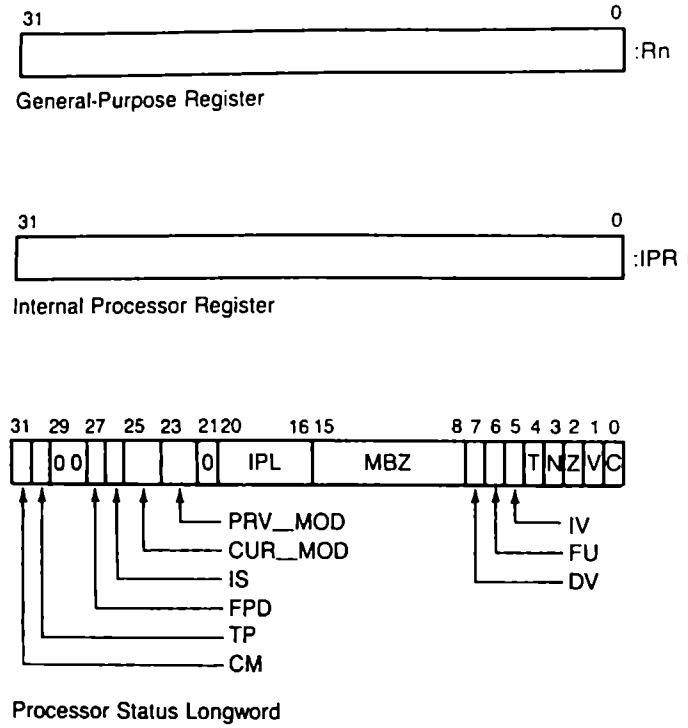


Figure 1.8
Processor State

- R12 is the argument pointer (AP). The VAX procedure call convention uses a data structure termed an argument list. AP contains the address of the base of this data structure.

Note that these registers are all used as base registers. The assignment of special meaning to these registers does not generally preclude their use for other purposes. As will be seen in Chapter 2, however, PC cannot be used as an accumulator, temporary, or index register.

When a datum of type byte, word, longword, or F_floating is stored in a register, the bit numbering in the register corresponds to the numbering in memory. Hence, a byte is stored in register bits ⟨7:0⟩, a word in register bits ⟨15:0⟩, and a longword or F_floating in register bits ⟨31:0⟩. Reading or writing a byte to a register accesses only bits ⟨7:0⟩ of the register; the other bits of the register are unaffected by writes and ignored for reads. Reading or writing a word to a register accesses only bits ⟨15:0⟩ of the register; the

other bits of the register are unaffected by writes and ignored for reads.

When a quadword, D_floating, or G_floating datum is stored in a register R[n], it is actually stored in two adjacent registers, R[n] and R[n+1]. Because of restrictions on the uses of PC (see Chapter 2), wraparound from PC to R0 and from SP to PC produces UNPREDICTABLE results. Bits ⟨31:0⟩ of the datum are stored in bits ⟨31:0⟩ of register R[n], and bits ⟨63:32⟩ of the datum are stored in bits ⟨31:0⟩ of register R[n+1].

When an octaword or H_floating datum is stored in register R[n], it is actually stored in adjacent registers R[n], R[n+1], R[n+2], and R[n+3]. Because of restrictions on the specification of PC (see Chapter 2), wraparound from PC to R0 and from AP, FP, and SP to PC produces UNPREDICTABLE results. Bits ⟨31:0⟩ of the datum are stored in bits ⟨31:0⟩ of register R[n], bits ⟨63:32⟩ in bits ⟨31:0⟩ of register R[n+1], bits ⟨95:64⟩ in bits ⟨31:0⟩ of register R[n+2], and bits ⟨127:96⟩ in bits ⟨31:0⟩ of register R[n+3].

A variable-length bit field may be specified in the registers with the restriction that the starting bit position P must be in the range 0 through 31. (see Figure 1.3). As for quadword, D_floating, and G_floating, a pair of registers R[n] and R[n+1] is treated as a 64-bit register with bits ⟨31:0⟩ in register R[n] and bits ⟨63:32⟩ in register R[n+1].

None of the string data types stored in registers can be processed by the VAX string instructions. Therefore, there is no architectural specification of the representation of strings in registers.

1.3.2 *Processor Status Longword*

The processor status longword (PSL) is a longword consisting of a word of privileged processor status concatenated with the processor status word (PSW), as shown in Figure 1.8 and described in Table 1.2. The processor status word contains the condition codes that give information on the results produced by previous VAX scalar instructions and the exception-enable bits that control the processor action on certain VAX scalar exception conditions. (See Chapter 5 for more about exceptions.) The condition codes are UNPREDICTABLE when they are affected by UNPREDICTABLE results. The VAX procedure call instructions conditionally set

Table 1.2
Fields of the Processor Status Longword (Part 1)

Extent	Name	Mnemonic	Meaning
⟨31⟩	Compatibility Mode	CM	When set, the processor is in PDP-11 compatibility mode (see Chapter 9). When CM is clear, the processor is in native mode. Compatibility mode may be omitted from implementations of the VAX architecture. In a processor that does not implement compatibility mode, this bit is always clear.
⟨30⟩	Trace Pending	TP	Forces a trace fault when set at the beginning of any instruction. Set by the processor if T is set at the beginning of an instruction.
⟨29:28⟩	Reserved		Reserved to DIGITAL; must be zero.
⟨27⟩	First Part Done	FPD	When set, execution of the instruction addressed by PC cannot simply be started at the beginning, and must be restarted at some other implementation-dependent point in its operation. If FPD is set and the exception or interrupt service routine modifies the general registers or the saved PSL (except for T or TP), the results of the restarted instruction's execution are UNPREDICTABLE. If a routine sets FPD, the results are also UNPREDICTABLE. However, if software is simulating unimplemented instructions, it may make free use of FPD in its simulation. If the hardware encounters a reserved instruction with FPD set, a reserved instruction fault is taken with the saved PSL⟨FPD⟩ set.

PSL⟨IV⟩ and PSL⟨DV⟩, clear PSL⟨FU⟩, and leave PSL⟨T⟩ unchanged at procedure entry. (See the descriptions of the procedure-call instructions in section 3.7 for more detail.)

The PSL is automatically saved on the stack when an exception or an interrupt occurs and is saved in the process control block on a process context switch. (See Chapters 5 and 6 for more detail.) The PSL can also be read by the MOVPSL instruction, described in section 3.8.

Bits ⟨31:16⟩ of the PSL can be changed explicitly only by executing a return from exception or interrupt (REI) instruction, described in section 5.12. Bits ⟨20:16⟩ can also be changed by a move-to-processor-register (MTPR) instruction to the IPL internal processor register, as described in Chapter 5. Processor initialization sets the PSL to 041F0000 (hex).

Table 1.2
Fields of the Processor Status Longword (Part 2)

Extent	Name	Mnemonic	Meaning
⟨26⟩	Interrupt Stack	IS	When set, the processor is executing on the interrupt stack. Any mechanism that sets IS also clears current mode and raises IPL above 0. If an REI attempts to restore a PSL with IS=1 and non-zero current mode or zero IPL, a reserved operand fault is taken. When clear, the processor is executing on the stack specified by current mode.
⟨25:24⟩	Current Access Mode	CUR_MOD	The access mode of the currently executing process: 0 = Kernel; 1 = Executive; 2 = Supervisor; 3 = User.
⟨22:23⟩	Previous Access Mode	PRV_MOD	Loaded from current mode by exceptions and CHMx instructions, cleared by interrupts, and restored by REI (see Chapter 5).
⟨21⟩	Reserved		Reserved to DIGITAL; must be zero.
⟨20:16⟩	Interrupt Priority Level	IPL	The current processor priority, in the range 0 through 1F (hex). The processor will accept interrupts only on levels greater than the current level.
⟨15:8⟩	Reserved		Reserved to DIGITAL; must be 0.
⟨7⟩	Decimal Overflow enable	DV	When set, forces a decimal overflow trap after execution of an instruction that produced an overflowed decimal result (no room to store a non-zero digit) or had a conversion error. When DV is clear, no trap occurs. (However, the condition code V bit is still set.)
⟨6⟩	Floating Underflow enable	FU	When set, forces a floating underflow exception after execution of a scalar instruction that produced an underflowed result. (That is, FU forces an exception when a result exponent, after normalization and rounding, is less than the smallest representable exponent for the data type.) When FU is clear, no exception occurs.

The vector processor does not use PSL⟨IV⟩ and PSL⟨FU⟩ to enable integer overflow and floating underflow exception checking for vector instructions. Also, vector instruction exceptions do not affect the values of the PSL condition codes bits.

1.3.3 *Internal Processor Registers*

The privileged internal processor register space provides access to many types of CPU control and status registers such as the memory management base registers, parts of the PSL, and the multiple stack pointers. These registers are explicitly accessible only by the Move to Processor Register (MTPR) and Move from Processor

Table 1.2
Fields of the Processor Status Longword (Part 3)

Extent	Name	Mnemonic	Meaning
⟨5⟩	Integer Overflow enable	IV	When set, forces an integer overflow trap after execution of a scalar instruction that produced an integer result that overflowed or had a conversion error. When IV is clear, no integer overflow trap occurs. (However, the condition code V bit is still set.)
⟨4⟩	Trace enable	T	When set at the beginning of an instruction, causes TP to be set. Most programs should treat T as UNPREDICTABLE because it is set by debuggers and trace programs for tracing and for proceeding from a breakpoint. See Chapter 5 for a description of tracing.
⟨3⟩	Negative	N	When set, indicates that the last instruction that affected N produced a result that was negative. When N is clear, the result was positive or 0.
⟨2⟩	Zero	Z	When set, indicates that the last instruction that affected Z produced a result that was 0. When Z is clear, the result was non-zero.
⟨1⟩	Overflow	V	When set, indicates that the last instruction that affected V produced a result whose magnitude was too large to be properly represented in the operand that received the result or there was a conversion error. When V is clear, there was no overflow or conversion error.
⟨0⟩	Carry	C	When set, indicates that the last instruction that affected C had a carry out of the most significant bit of the result or a borrow into tne most significant bit. When C is clear, there was no carry or borrow.

Register (MFPR) instructions, which require kernel mode privileges. Internal processor registers are longword size, as shown in Figure 1.8. Internal processor registers and the MTPR and MFPR instructions are described in Chapter 8.

2

Instruction Formats and Addressing Modes

The VAX architecture has a variable-length instruction format. An instruction specifies an operation and 0 to 6 operands. An operand specifier determines how an operand is accessed. An operand specifier consists of an addressing mode specifier and, if needed, a specifier extension, immediate data, or an address, as shown in Figure 2.1. The format of an instruction is as follows:

```
opcode
addressing mode specifier 1 (if needed)
specifier extension, address, or immediate data 1 (if needed)
addressing mode specifier 2 (if needed)
   .
   .
   .
addressing mode specifier n (if needed)
specifier extension, address, or immediate data n (if needed)
```

2.1 Opcode Formats

An instruction is specified by the byte address A of its opcode, as shown in Figure 2.1. The opcode may extend over 2 bytes; the length depends on the contents of the byte at address A. Only if the value of the byte is FC (hex) through FF (hex) is the opcode 2 bytes long, as shown in Figure 2.1.

2.2 Operand Specifiers

Each instruction takes a specific sequence of operand specifier types. An operand specifier type conceptually has two attributes: the access type and the data type.

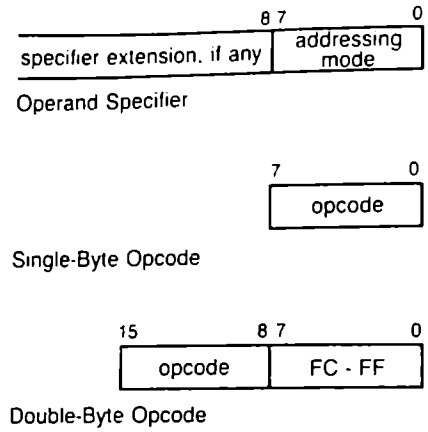


Figure 2.1
Opcodes and Operand Specifiers

The access types include the following:

- Read—the specified operand is read only.
- Write—the specified operand is written only.
- Modify—the specified operand is read, potentially modified, and written. This is not done under a memory interlock.
- Address—the address of the specified operand in the form of a longword is the actual instruction operand. The specified operand is not accessed directly although the instruction may subsequently use the address to access that operand.
- Variable-length bit field base address—this is the same as address access type except for register mode. In register mode, the field is contained in register n designated by the operand specifier (or register n+1 concatenated with register n). This access type is a special variant of the address access type.
- Branch—no operand is accessed. The operand specifier itself is a branch displacement.

The first five types are termed general mode addressing. The last type is termed branch mode addressing.

The data types include the following:

- Byte
- Word
- Longword
- Quadword

- Octaword
- F_floating
- D_floating
- G_floating
- H_floating

The address and branch access types do not directly reference operands. For the address access type, the data type indicates the operand size to be used in the address calculation in autoincrement, autodecrement, and index modes. For the branch access type, the data type indicates the size of the branch displacement.

2.3 Notation

To describe the addressing modes, the following notation is used:

+	Addition
–	Subtraction
*	Multiplication
←	Is replaced by
=	Is defined as
'	Concatenation
Rn or R[n]	The contents of register n
PC or SP	The contents of register 15 or 14, respectively
(x)	The contents of memory location x
( )	Arithmetic parentheses for indicating precedence
SEXT(x)	Sign extend x to size of operand needed
ZEXT(x)	Zero extend x to size of operand needed
OA	Operand address
!	Comment delimiter

Each general mode addressing description includes the definition of the operand address and the specified operand. For operand specifiers of address access type, the operand address is the actual instruction operand; for other access types, the specified operand is the instruction operand. The branch mode addressing description includes the definition of the branch address.

2.4 General Mode Addressing Formats

Except for literal mode, an operand specifier in the general mode addressing format consists of a register number in bits ⟨3:0⟩ and

an addressing mode specifier in bits ⟨7:4⟩. The operand specifier could be followed by a specifier extension, as shown in Figure 2.1.

For a summary of general register addressing, see Table 2.1.

2.4.1 *Register Mode*

The register mode operand specifier format is shown in Figure 2.2 (see p. 32). No specifier extension follows. In register mode addressing, the operand is the contents of register n (or register n+1 concatenated with register n for quadword, D_floating, G_floating, and certain field operands). The format is as follows:

```
operand = Rn                          ! if one register
          or
          R[n+1]'Rn                   ! if two registers
          or
          R[n+3]'R[n+2]'R[n+1]'Rn     ! if four registers
```

Because registers do not have memory addresses, the operand address is not defined and register mode may not be used for operand specifiers of address access type (except in the case of the base address for bit-field instructions; see section 3.5). If register mode is so used, an illegal addressing mode fault results. (See Chapter 5 for a description of faults.) PC may not be used in register mode addressing. If PC is read, the value read is UNPREDICTABLE. If PC is written, the next instruction executed or the next operand specified is UNPREDICTABLE. Likewise, SP may not be used in register mode addressing for an operand that takes two adjacent registers. Again, if it is used, the results are UNPREDICTABLE in the same fashion. If PC is used in register mode for a write access type operand that takes two adjacent registers, the contents of R0 are UNPREDICTABLE. If R12, R13, SP, or PC is used in register mode addressing for an operand that takes four adjacent registers, the results are UNPREDICTABLE. If PC is used in register mode for a write access type operand that requires four adjacent registers, the contents of R0, R1, and R2 are UNPREDICTABLE. Likewise, if R13 is used in register mode for a write access type operand that takes four adjacent registers, the contents of R0 are UNPREDICTABLE; if SP is used in register mode for a write access type operand that takes four adjacent registers, the contents of R0 and R1 are UNPREDICTABLE.

The assembler notation for register mode is Rn.

2.4.2 *Register Deferred Mode*

The register deferred mode operand specifier format is shown in Figure 2.2. No specifier extension follows. In register deferred mode addressing, the address of the operand is the contents of register n:

```
OA = Rn
operand = (OA)
```

PC should not be used in register deferred mode addressing. If PC is used, the address of the operand (and whether the operand is written if it is of modify or write access type) is UNPREDICTABLE.

The assembler notation for register deferred mode is (Rn).

2.4.3 *Autoincrement Mode*

The autoincrement mode operand specifier format is shown in Figure 2.2. No specifier extension follows. If Rn denotes PC, an immediate datum follows, and the mode is termed immediate mode. See Figure 2.2. In autoincrement mode addressing, the address of the operand is the contents of register n. After the operand address is determined, the size of the operand in bytes (1 for byte; 2 for word; 4 for longword or F_floating; 8 for quadword, G_floating, and D_floating; and 16 for octaword and H_floating) is added to the contents of register n. The contents of register n are then replaced by the result:

```
OA = Rn
Rn ← Rn + size
operand = (OA)
```

Immediate mode may not be used for operands of modify or write access type. If immediate mode is used for an operand of modify access type, the value of the data read is UNPREDICTABLE. If immediate mode is used for an operand of modify or write access type, the address at which the operand is written (and whether it is written) is UNPREDICTABLE.

The assembler notation for autoincrement mode is (Rn)+. For immediate mode, the notation is I^#constant, where constant is the immediate datum that follows.

Table 2.1
Summary of General Register Addressing

Addressing Mode	Assembler Notation	Decimal	Hexadecimal
General register addressing mode			
literal	S^#literal	0–3	0–3
indexed	base[Rx][a]	4	4
register	Rn	5	5
register deferred	(Rn)	6	6
autodecrement	-(Rn)	7	7
autoincrement	(Rn)+	8	8
autoincrement deferred	@(Rn)+	9	9
byte displacement	B^displacement(Rn)	10	A
byte displacement deferred	@B^displacement(Rn)	11	B
word displacement	W^displacement(Rn)	12	C
word displacement deferred	@W^displacement(Rn)	13	D
longword displacement	L^displacement(Rn)	14	E
longword displacement deferred	@L^displacement(Rn)	15	F
Program counter addressing mode			
immediate	I^#constant	8	8
absolute	@#address	9	9
byte relative	B^address	10	A
byte relative deferred	@B^address	11	B
word relative	W^address	12	C
word relative deferred	@W^address	13	D
longword relative	L^address	14	E
longword relative deferred	@L^address	15	F

[a]The word "base" indicates any indexable addressing mode.
[b]The column headings show read, modify, write, address, and field access, respectively.
[c]The codes in these columns have the following meanings:

- f reserved addressing mode fault
- p Program Counter addressing
- u UNPREDICTABLE
- ud UNPREDICTABLE for destination of CALLS, CALLG, JMP, and JSB
- uq UNPREDICTABLE for quadword, octaword, D_floating, G_floating, and H_floating (and field if position plus size greater than 32)
- uo UNPREDICTABLE for octaword and H_floating
- ux UNPREDICTABLE for index register same as base register
- y yes, always valid addressing mode

Table 2.1
Summary of General Register Addressing

Access[b,c]								
r	m	w	a	v	PC[c]	SP[c]	AP & FP[c]	Indexable[c]
y	f	f	f	f				f
y	y	y	y	y	f	y	y	f
y	y	y	f	y	u	uq	uo	f
y	y	y	y	y	u	y	y	y
y	y	y	y	y	u	y	y	ux
y	y	y	y	y	p	y	y	ux
y	y	y	y	y	p	y	y	ux
y	y	y	y	y	p	y	y	y
y	y	y	y	y	p	y	y	y
y	y	y	y	y	p	y	y	y
y	y	y	y	y	p	y	y	y
y	y	y	y	y	p	y	y	y
y	y	y	y	y	p	y	y	y
y	u	u	y	y				u
y	y	y	y	y				y
y	y	y	y	y				y
y	y	y	y	y				y
y	y	y	y	y				y
y	y	y	y	y				y
y	y	y	y	y				y
y	y	y	y	y				y

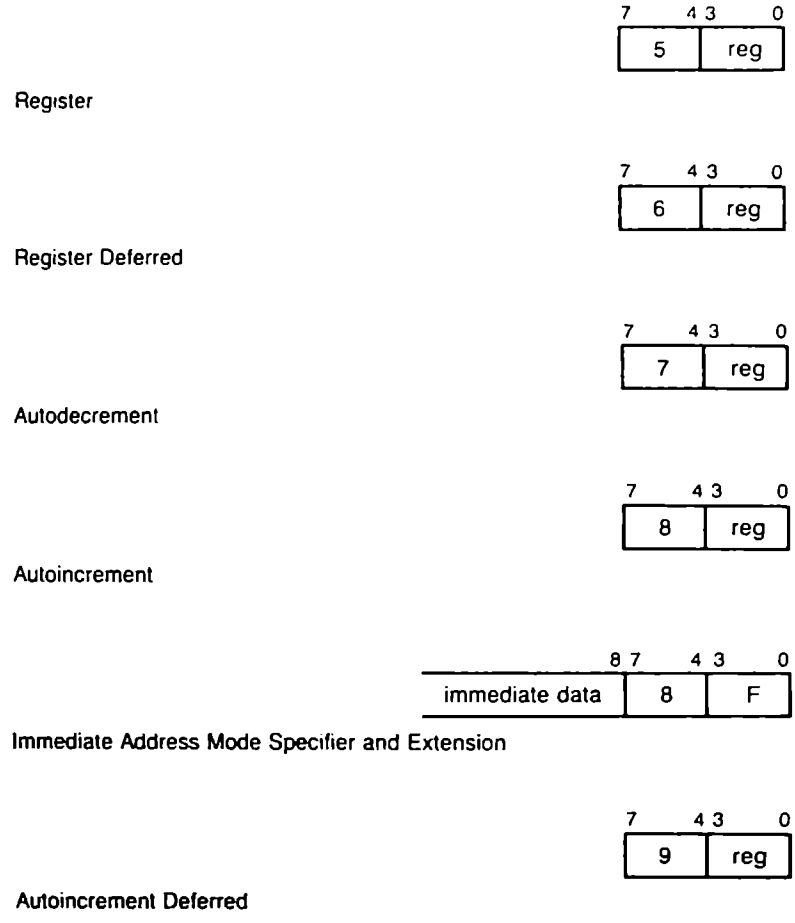
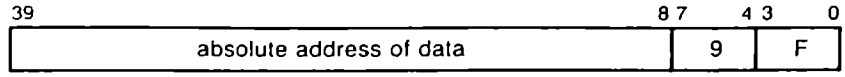


Figure 2.2
Addressing-Mode Specifiers (Part 1)

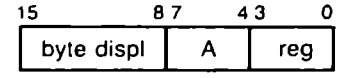
2.4.4 *Autoincrement Deferred Mode*

The autoincrement deferred mode operand specifier format appears in Figure 2.2. No specifier extension follows. If Rn denotes PC, a longword address follows, and the mode is termed absolute mode. See Figure 2.2. In autoincrement deferred mode addressing, the address of the operand is the contents of a longword whose address is the contents of register n. After the operand address is determined, 4 (the size in bytes of a longword address) is added to the contents of register n and the contents of register n are replaced by the result:

```
OA = (Rn)
Rn ← Rn + 4
```

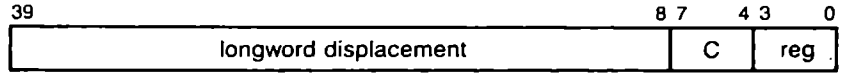


Absolute Address Mode Specifier and Extension



Byte Displacement Address Mode Specifier and Extension

Word Displacement Address Mode Specifier and Extension



Longword Displacement Address Mode Specifier and Extension

Figure 2.2
Addressing Mode Specifiers (Part 2)

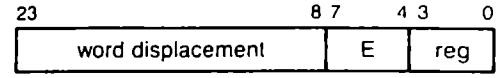
```
operand = (OA)
```

The assembler notation for autoincrement deferred mode is @(Rn)+. The notation for absolute mode is @#address, where address is the longword that follows.

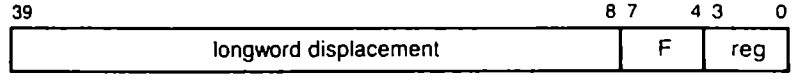
2.4.5 *Autodecrement Mode*

The autodecrement mode operand specifier format is shown in Figure 2.2. No specifier extension follows. In autodecrement mode addressing, the size of the operand in bytes (1 for byte; 2 for word; 4 for longword and F_floating; 8 for quadword, G_floating, and D_floating; and 16 for octaword and H_floating) is subtracted from the contents of register n. The contents of register n are then replaced by the result. The updated contents of register n are the address of the operand:

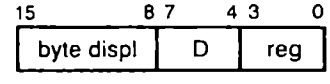
```
Rn ← Rn - size
OA = Rn
```



Word Displacement Deferred Address Mode Specifier and Extension



Longword Displacement Deferred Address Mode Specifier and Extension



Byte Displacement Deferred Address Mode Specifier and Extension

Figure 2.2
Addressing Mode Specifiers (Part 3)

```
operand = (OA)
```

PC should not be used in autodecrement mode. If it is used, the address of the operand (and whether the operand is written if it is of modify or write access type) is UNPREDICTABLE; the next instruction executed or the next operand specified is also UNPREDICTABLE.

The assembler notation for autodecrement mode is –(Rn).

2.4.6 *Displacement Mode*

There are three displacement mode operand specifier formats. They are termed byte displacement mode, word displacement mode, and longword displacement mode. In each, the specifier extension is a signed displacement. See Figure 2.2.

In displacement mode addressing, the displacement (after being sign extended to 32 bits if it is byte or word) is added to the contents of register n. The result is the operand address:

```
OA = Rn + SEXT(displ)    ! if byte or word displacement
     or
     Rn + displ          ! if longword displacement
operand = (OA)
```

If Rn denotes PC, the mode is termed PC relative addressing mode. The updated contents of PC (the address of the first byte beyond the specifier extension) are used as the base address.

The assembler notations for byte, word, and longword displacement mode are B^D(Rn), W^D(Rn), and L^D(Rn), respectively, with D representing displ.

2.4.7 *Displacement Deferred Mode*

The three displacement deferred mode operand specifier formats are termed byte displacement deferred mode, word displacement deferred mode, and longword displacement deferred mode. In each, the specifier extension is a signed displacement. See Figure 2.2.

In displacement deferred mode addressing, the displacement (after being sign extended to 32 bits if it is byte or word) is added to the contents of register n. The result is the address of a longword whose contents are the operand address:

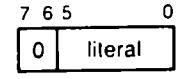
```
OA = (Rn + SEXT(displ))    ! if byte or word displacement
     or
     (Rn + displ)          ! if longword displacement
operand = (OA)
```

If Rn denotes PC, the mode is termed PC relative deferred addressing mode. The updated value of PC (the address of the first byte beyond the specifier extension) is used as the base address.

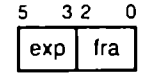
The assembler notations for byte, word, and longword displacement deferred mode are @B^D(Rn), @W^D(Rn), and @L^D(Rn), respectively, with D representing displ.

2.4.8 *Literal Mode*

The literal mode operand specifier format is shown in Figure 2.3. No specifier extension follows. For operands of data type byte,



Literal Address Mode Specifier



Representation of a Floating-Point Number as a Literal

Figure 2.3
Literal Address Mode

word, longword, quadword, and octaword, the operand is the zero extension of the 6-bit literal field:

```
operand = ZEXT(literal)
```

Thus, for these data types, literal mode may be used for values in the range 0 through 63.

For operands of data type F_floating, D_floating, G_floating, and H_floating, the 6-bit literal field is composed of two 3-bit fields, as shown in Figure 2.3. The exp and fra fields are used to form an F_floating, D_floating, G_floating, or H_floating operand, as shown in Figure 2.4. The values that can be expressed by a floating-point literal are shown in Table 2.2.

Because there is no operand address, literal mode addressing may not be used for operand specifiers of address access type. Also, literal mode addressing may not be used for operand specifiers of write or modify access type. If literal mode is used for operand specifiers of address, modify, or write access type, an illegal addressing mode fault results. (See Chapter 5 for a description of faults.)

Literal mode addressing is a very efficient way of specifying integer values in the range 0 through 63 or the floating-point values shown in Table 2.2. Literal values outside the indicated range may be obtained by using immediate mode.

The assembler notation for literal mode is S^#literal.

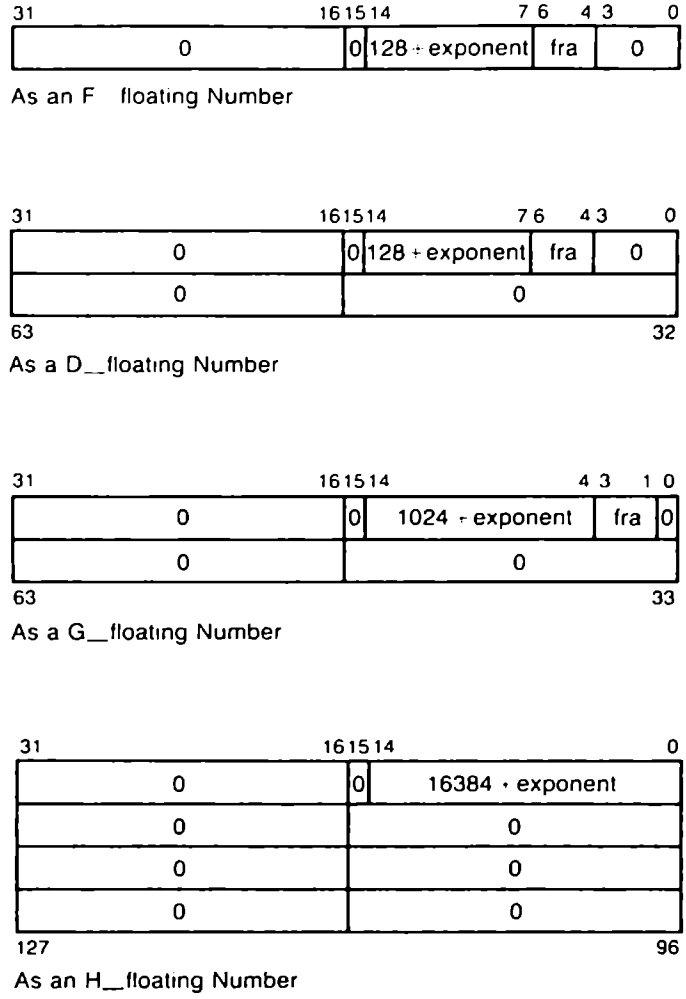


Figure 2.4
Interpretation of a Literal

2.4.9 *Index Mode*

The index mode operand specifier format is shown in Figure 2.5. Bits ⟨15:8⟩ contain a second operand specifier (termed the base operand specifier) for any of the addressing modes except register, literal, or index. The specification of register indexed, literal indexed, immediate indexed, or index indexed addressing mode results in an illegal addressing mode fault (described in Chapter 5). If the base operand specifier requires a specifier extension, it immediately follows. The base operand specifier is subject to the same restrictions as would apply if it were used alone. If the use of

Table 2.2
Floating-Point Values Representable as Literals

	Fraction							
Exponent	0	1	2	3	4	5	6	7
0	1/2	9/16	5/8	11/16	3/4	13/16	7/8	15/16
1	1	1 1/8	1 1/4	1 3/8	1 1/2	1 5/8	1 3/4	1 7/8
2	2	2 1/4	2 1/2	2 3/4	3	3 1/4	3 1/2	3 3/4
3	4	4 1/2	5	5 1/2	6	6 1/2	7	7 1/2
4	8	9	10	11	12	13	14	15
5	16	18	20	22	24	26	28	30
6	32	36	40	44	48	52	56	60
7	64	72	80	88	96	104	112	120

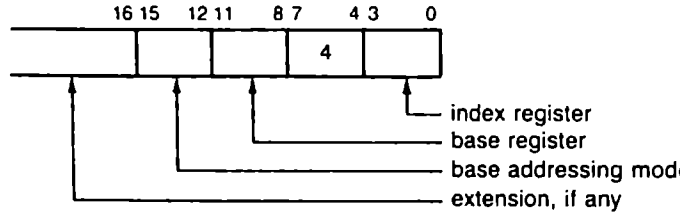


Figure 2.5
Indexed Specifier and Extension

some particular specifier is illegal (causes a fault or UNPREDICTABLE behavior) under some circumstances, then that specifier is similarly illegal as a base operand specifier in index mode under the same circumstances.

The operand to be specified by index mode addressing is termed the primary operand. The base operand specifier normally is used to determine an operand address. This address is termed the base operand address (BOA). The address of the primary operand specified is determined by multiplying the contents of the index register x by the size of the primary operand in bytes (1 for byte; 2 for word; 4 for longword and F_floating; 8 for quadword, D_floating, and G_floating; and 16 for octaword and H_floating), adding BOA, and taking the result:

```
OA = BOA + size * Rx
```

```
operand = (OA)
```

If the base operand specifier is for autoincrement or autodecrement mode, the increment or decrement size is the size in bytes of the primary operand.

Indexed mode addressing permits very general and efficient accessing of arrays. The base address of the array is determined by the operand address calculation of the base operand specifier. The contents of the index register are taken as a logical index into the array. The logical index is converted into a real (byte) offset by multiplying the contents of the index register by the size of the primary operand in bytes.

Certain restrictions are placed on the index register x. PC cannot be used as an index register. If PC is used, a reserved addressing mode fault occurs (described in Chapter 5). If the base operand specifier is for an addressing mode that results in register modification (autoincrement mode, autodecrement mode, or autoincrement deferred mode), the same register cannot be the index register. If it is, the primary operand address is UNPREDICTABLE.

The names of the addressing modes resulting from indexed mode addressing are formed by appending the word "indexed" to the addressing mode of the base operand specifier. Following are the names and assembler notations. The index register is designated Rx to distinguish it from the register Rn used in the base operand specifier.

- Register deferred indexed, (Rn)[Rx]
- Autoincrement indexed, (Rn)+[Rx]
- Autoincrement deferred indexed, @(Rn)+[Rx], or absolute indexed, @#address[Rx]
- Autodecrement indexed, –(Rn)[Rx]
- Byte, word, or longword displacement indexed, B^D(Rn)[Rx], W^D(Rn)[Rx], or L^D(Rn)[Rx]
- Byte, word, or longword displacement deferred indexed, @B^D(Rn)[Rx], @W^D(Rn)[Rx], or @L^D(Rn)[Rx]

2.5 *Branch Mode Addressing Formats*

The two operand specifier formats are shown in Figure 2.6. In branch mode addressing, the byte or word displacement is sign

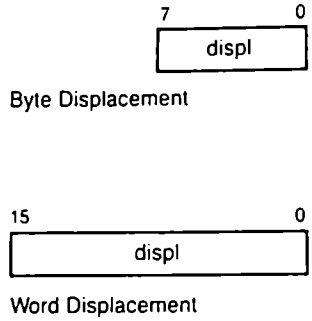


Figure 2.6
Branch-Mode Specifiers

extended to 32 bits and added to the updated contents of PC. The updated contents of PC are the address of the first byte beyond the operand specifier. The result is the branch address A:

```
A = PC + SEXT(displ)
```

The assembler notation for byte and word branch mode addressing is A, where A is the branch address. Note that the branch address and not the displacement is used.

2.6 *Instruction Interpretation*

The VAX architecture assumes a sequential control-flow model of instruction stream processing. In this model, instructions are processed from the instruction stream, which resides in instruction memory, one at a time, and no instruction is processed (even partially) until the one before it completes.

In interpreting an instruction, the processor performs the following three steps:

1. Reads and evaluates each operand specifier in order of instruction stream occurrence as follows:
   a. If access type is read: evaluates the operand address; if the operand is in a register, reads the operand and saves it; if the operand is not in a register, does the reading and saving of the operand anytime before performing step 2.
   b. If access type is write: evaluates the operand address and saves it.

c. If access type is modify: evaluates the operand address and saves it; if the operand is in a register, reads the operand and saves it; if the operand is not in a register, does the reading and saving of the operand anytime before performing step 2.
d. If access type is address: evaluates the address and saves it.
e. If access type is branch: saves the operand specifier.

The read accesses for operands in registers are done as the operand specifiers are processed. The bytes for non-register operands may be read in any order anytime before performing step 2. (The order in which one processor reads bytes for non-register operands within a single instruction may differ from the order in which a second processor wrote them.

2. Performs the operation indicated by the instruction.
3. Stores the result(s) using the saved addresses.

   Within an operand, the result bytes may be stored in any order. Also, the result operands may be stored in any order with one restriction: if multiple results overlap, the order of operand storage is indicated by the occurrence of the corresponding operand specifiers in the instruction stream. For example, if an instruction references two operands of write or modify access type at the same address, the first will be overwritten by the second.

The implications of this instruction interpretation process are as follows:

- Autoincrement and autodecrement operations occur as the operand specifiers are processed, and subsequent operand specifiers use the updated contents of registers modified by those operations.
- Other than as indicated in the note below, all input operands are read, and all addresses of output operands computed before any results of the instruction are stored.
- An operand of modify access type is not read, modified, and written as an indivisible operation; therefore, modify-access-type operands cannot be used for multiprocessor synchronization (see section 7.1.2).

Note
The string instructions are an exception to this sequence performed by each instruction. Partial results are stored before the instruction operation is completed.

The variable-length bit field instructions treat the position, size, and base address operand specifiers as the specification of an implied field operand specifier.

If multiple exceptions occur, the order in which they are taken is UNPREDICTABLE (except for the trace fault—see section 5.7.5). This can occur, for example, in a floating-point instruction whose destination operand specifier of write access type uses a reserved addressing mode and the operation results in an overflow fault.

3

Instructions

This chapter describes the instructions generally used by all software across all implementations of the VAX architecture. Certain instructions are specific to portions of the VAX architecture: memory management, interrupts and exceptions, process dispatching, and processor registers. Such instructions are generally used by privileged software and are described in chapters devoted to those portions of the architecture. A concise list of opcode assignments appears in Appendix A.

The instruction set is divided into 12 major sections:

- Integer-arithmetic and logical
- Address
- Variable-length bit field
- Control
- Procedure call
- Miscellaneous
- Queue
- Floating point
- Character string
- Cyclic redundancy check
- Decimal string
- Edit

Within each section, closely related instructions are grouped and described together. The instruction group description is composed of the following:

1. The group name.
2. The format of each instruction in the group. The format presents the name and the type of each instruction operand specifier and the order in which it appears in memory. Operand specifiers from left to right appear in increasing memory addresses. Implied operands are enclosed by { }.
3. The operation of the instruction. The operation is given as a sequence of pseudocode statements in an ALGOL-like syntax. Each VAX processor may implement the instruction in different or more efficient ways, but each processor gives results consistent with the pseudocode, English descriptions, and notes.
4. The effect on condition codes.
5. Exceptions specific to the instruction. Exceptions generally possible for all instructions are not listed (for example, illegal or reserved addressing mode, trace, and memory management exceptions).
6. The opcodes, mnemonics, and names of each instruction in the group. The opcodes are given in hex.
7. A description in English of the instruction.
8. In many cases, notes on the instruction and programming examples.

3.1 *Operand Specifier Notation*

Operand specifiers are described in the following way:

```
<name>.<access type><data type>
```

The name is suggestive of the operand in the context of the instruction. The name is often abbreviated. The access type is represented by a letter denoting the operand specifier access type. These are as follows:

a Calculate the effective address of the specified operand. Address is returned in a longword that is the actual instruction operand. Context of address calculation (the size to be used in autoincrement, autodecrement, and indexing) is given by ⟨data type⟩.

b No operand reference. The operand specifier is a branch displacement. The size of branch displacement is given by ⟨data type⟩.

m Operand is read, potentially modified, and written. Note that this is not an indivisible memory operation. Also note that if the operand is not actually modified, it may not be written back. However, modify type operands are always checked for both read and write accessibility (see Chapter 4).

r Operand is read only.

v Calculate the effective address of the specified operand. If the effective address is in memory, the address is returned in a longword that is the actual instruction operand. The context of the address calculation is given by ⟨data type⟩. If the effective address is Rn, the operand is in Rn or R[n+1]'Rn.

w Operand is written only.

The data type in the operand specifier notation is a letter denoting the data type of the operand:

b Byte

d D_floating

f F_floating

g G_floating

h H_floating

l Longword

o Octaword

q Quadword

w Word

x First data type specified by instruction

y Second data type specified by instruction

• Multiple longwords

3.2 *Operation Description Notation*

The operation of each instruction is given as a sequence of control and assignment statements in an ALGOL-like syntax. No attempt is made to define the syntax formally; it is assumed to be familiar to the reader. The notation used is an extension of that introduced in Chapter 2.

+	Addition
–	Subtraction, unary minus
•	Multiplication
/	Division (quotient only)

**	Exponentiation
'	Concatenation
←	Is replaced by
=	Is defined as
Rn or R[n]	Contents of register Rn
PC, SP, FP, or AP	The contents of register R15, R14, R13, or R12, respectively
PSW	The contents of the processor status word
PSL	The contents of the processor status longword
(x)	Contents of memory location whose address is x
(x)+	Contents of memory location whose address is x; x incremented by the size of operand referenced at x
-(x)	Address x decremented by size of operand to be referenced at x; contents of memory location whose address is new value of x
⟨x:y⟩	A modifier that delimits an extent from bit position x to bit position y inclusive
⟨x1,x2,...,xn⟩	A modifier that enumerates bits x1,x2,...,xn
( )	Arithmetic parentheses used to indicate precedence
AND	Logical AND
OR	Logical OR
XOR	Logical XOR
NOT	Logical (one's) complement
LSS	Less than signed
LSSU	Less than unsigned
LEQ	Less than or equal signed
LEQU	Less than or equal unsigned
EQL	Equal signed
EQLU	Equal unsigned
NEQ	Not equal signed
NEQU	Not equal unsigned
GEQ	Greater than or equal signed
GEQU	Greater than or equal unsigned
GTR	Greater than signed
GTRU	Greater than unsigned
SEXT(x)	Sign extend x to size of operand needed
ZEXT(x)	Zero extend x to size of operand needed

REM(x,y)	Remainder of x divided by y, such that x/y and REM(x,y) have the same sign
MINU(x,y)	Minimum unsigned of x and y
MAXU(x,y)	Maximum unsigned of x and y

The following conventions are used:

1. Other than that caused by ( )+ or −( ), and the advancement of PC, only operands or portions of operands appearing on the left side of assignment statements are affected.
2. No operator precedence is assumed other than that replacement (←) has the lowest precedence. Precedence is indicated explicitly by ( ).
3. All arithmetic, logical, and relational operators are defined in the context of their operands. For example, "+" applied to floating operands means a floating add, whereas "+" applied to byte operands is an integer byte add. Similarly, "LSS" is a floating comparison when applied to floating operands, whereas "LSS" is an integer byte comparison when applied to byte operands.
4. Instruction operands are evaluated according to the operand specifier conventions (see the description of instruction interpretation in section 2.6). The order in which operands appear in the instruction description has no effect on the order of evaluation.
5. Condition codes are affected in general on the value of actual stored results, not on "true" results (which might be generated internally to greater precision). Thus, for example, two positive integers can be added together and the sum stored, because of overflow, as a negative value. The condition codes will indicate a negative value even though the "true" result is clearly positive.

3.3 *Integer-Arithmetic and Logical Instructions*

This class of instructions performs integer and logical operations on byte, word, longword, quadword, and octaword data types. VAX processors are designed to optimize the performance of these instructions for all data types except octaword. The instructions that operate on octawords belong to an instruction group that is optional to implement; refer to Chapter 11.

ADAWI Add Aligned Word Interlocked

Format

```
opcode add.rw, sum.mw
```

Operation

```
tmp ← add;
{set interlock};
sum ← sum + tmp;
{release interlock};
```

Condition codes

```
N ← sum LSS 0;
Z ← sum EQL 0;
V ← {integer overflow};
C ← {carry from most significant bit};
```

Exceptions

```
reserved operand fault
integer overflow
```

Opcodes

58 ADAWI Add Aligned Word Interlocked

Description

The addend operand is added to the sum operand, and the sum operand is replaced by the result. If the sum operand is contained in memory, then the operation is interlocked against interlocked operations to the same address from other processors. The destination must be aligned on a word boundary (bit 0 of the address of the sum operand must be zero). If it is not, a reserved operand fault is taken.

Notes

1. Integer overflow occurs if the input operands to the add have the same sign and the result has the opposite sign. On overflow, the sum operand is replaced by the low order bits of the true result.
2. If the addend and the sum operands overlap, the result and the condition codes are UNPREDICTABLE.

ADD Add

Format

```
opcode add.rx, sum.mx                2 operand
opcode add1.rx, add2.rx, sum.wx      3 operand
```

Operation

```
sum ← sum + add;                     !2 operand
sum ← add1 + add2;                   !3 operand
```

Condition codes

```
N ← sum LSS 0;
Z ← sum EQL 0;
V ← {integer overflow};
C ← {carry from most significant bit};
```

Exceptions

```
integer overflow
```

Opcodes

80	ADDB2	Add Byte 2 Operand
81	ADDB3	Add Byte 3 Operand
A0	ADDW2	Add Word 2 Operand
A1	ADDW3	Add Word 3 Operand
C0	ADDL2	Add Long 2 Operand
C1	ADDL3	Add Long 3 Operand

Description

In 2 operand format, the addend operand is added to the sum operand and the sum operand is replaced by the result. In 3 operand format, the addend 1 operand is added to the addend 2 operand and the sum operand is replaced by the result.

Notes

1. Integer overflow occurs if the input operands to the add have the same sign and the result has the opposite sign. On overflow, the sum operand is replaced by the low order bits of the true result.

ADWC Add With Carry

Format

```
opcode add.rl, sum.ml
```

Operation

```
sum ← sum + add + C;
```

Condition codes

```
N ← sum LSS 0;
Z ← sum EQL 0;
V ← {integer overflow};
C ← {carry from most significant bit};
```

Exceptions

```
integer overflow
```

Opcodes

D8 ADWC Add With Carry

Description

The contents of PSL⟨C⟩ and the addend operand are added to the sum operand, and the sum operand is replaced by the result.

Notes

1. On overflow, the sum operand is replaced by the low order bits of the true result.
2. The two additions required by this operation are performed simultaneously.

ASH — Arithmetic Shift

Format

```
opcode cnt.rb, src.rx, dst.wx
```

Operation

```
dst ← src shifted cnt bits;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {integer overflow};
C ← 0;
```

Exceptions

```
integer overflow
```

Opcodes

78	ASHL	Arithmetic Shift Long
79	ASHQ	Arithmetic Shift Quad

Description

The source operand is arithmetically shifted by the number of bits specified by the count operand, and the destination operand is replaced by the result. The source operand is unaffected. A positive count operand shifts to the left, bringing zeros into the least significant bit. A negative count operand shifts to the right, bringing copies of the most significant (sign) bit into the most significant bit. A 0 count operand replaces the destination operand with the unshifted source operand.

Notes

1. Integer overflow occurs on a left shift if any bit shifted into the sign bit position differs from the sign bit of the source operand.
2. If cnt GTR 32 (ASHL) or cnt GTR 64 (ASHQ), the destination operand is replaced by 0.
3. If cnt LEQ −31 (ASHL) or cnt LEQ −63 (ASHQ), all the bits of the destination operand are copies of the sign bit of the source operand.

BIC — Bit Clear

Format

```
opcode mask.rx, dst.mx                2 operand
opcode mask.rx, src.rx, dst.wx        3 operand
```

Operation

```
dst ← dst AND {NOT mask};             !2 operand
dst ← src AND {NOT mask};             !3 operand
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

8A	BICB2	Bit Clear Byte
8B	BICB3	Bit Clear Byte
AA	BICW2	Bit Clear Word
AB	BICW3	Bit Clear Word
CA	BICL2	Bit Clear Long
CB	BICL3	Bit Clear Long

Description

In 2 operand format, the destination operand is ANDed with the one's complement of the mask operand, and the destination operand is replaced by the result. In 3 operand format, the source operand is ANDed with the one's complement of the mask operand, and the destination operand is replaced by the result.

BIS — Bit Set

Format

```
opcode mask.rx, dst.mx                2 operand
opcode mask.rx, src.rx, dst.wx        3 operand
```

Operation

```
dst ← dst OR mask;                    !2 operand
dst ← src OR mask;                    !3 operand
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

88	BISB2	Bit Set Byte 2 Operand
89	BISB3	Bit Set Byte 3 Operand
A8	BISW2	Bit Set Word 2 Operand
A9	BISW3	Bit Set Word 3 Operand
C8	BISL2	Bit Set Long 2 Operand
C9	BISL3	Bit Set Long 3 Operand

Description

In 2 operand format, the mask operand is ORed with the destination operand, and the destination operand is replaced by the result. In 3 operand format, the mask operand is ORed with the source operand, and the destination operand is replaced by the result.

BIT — Bit Test

Format

```
opcode mask.rx, src.rx
```

Operation

```
tmp ← src AND mask;
```

Condition codes

```
N ← tmp LSS 0;
Z ← tmp EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

93	BITB	Bit Test Byte
B3	BITW	Bit Test Word
D3	BITL	Bit Test Long

Description

The mask operand is ANDed with the source operand. Both operands are unaffected. The only action is to affect condition codes.

CLR — Clear

Format

```
opcode dst.wx
```

Operation

```
dst ← 0;
```

Condition codes

```
N ← 0;
Z ← 1;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

94	CLRB	Clear Byte
B4	CLRW	Clear Word
D4	CLRL	Clear Long
7C	CLRQ	Clear Quad
7CFD	CLRO	Clear Octa

Description

The destination operand is replaced by 0.

Notes

1. *CLRx dst* is equivalent to *MOVx S^#0, dst*, but is 1 byte shorter.
2. The CLRO instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

CMP — Compare

Format

```
opcode src1.rx, src2.rx
```

Operation

```
src1 - src2;
```

Condition codes

```
N ← src1 LSS src2;
Z ← src1 EQL src2;
V ← 0;
C ← src1 LSSU src2;
```

Exceptions

none

Opcodes

91	CMPB	Compare Byte
B1	CMPW	Compare Word
D1	CMPL	Compare Long

Description

The source 1 operand is compared with the source 2 operand. The only action is to affect the condition codes.

CVT — Convert

Format

```
opcode src.rx, dst.wy
```

Operation

```
dst ← conversion of src;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {integer overflow};
C ← 0;
```

Exceptions

```
integer overflow
```

Opcodes

99	CVTBW	Convert Byte to Word
98	CVTBL	Convert Byte to Long
33	CVTWB	Convert Word to Byte
32	CVTWL	Convert Word to Long
F6	CVTLB	Convert Long to Byte
F7	CVTLW	Convert Long to Word

Description

The source operand is converted to the data type of the destination operand, and the destination operand is replaced by the result. Conversion of a shorter data type to a longer one is done by sign extension; conversion from longer to shorter is done by truncation of the higher numbered (most significant) bits.

Notes

1. Integer overflow occurs if any truncated bits of the source operand are not equal to the sign bit of the destination operand.

DEC — Decrement

Format

```
opcode dif.mx
```

Operation

```
dif ← dif - 1;
```

Condition codes

```
N ← dif LSS 0;
Z ← dif EQL 0;
V ← {integer overflow};
C ← {borrow into most significant bit};
```

Exceptions

```
integer overflow
```

Opcodes

97	DECB	Decrement Byte
B7	DECW	Decrement Word
D7	DECL	Decrement Long

Description

One is subtracted from the difference operand, and the difference operand is replaced by the result.

Notes

1. Integer overflow occurs if the largest negative integer is decremented. On overflow, the difference operand is replaced by the largest positive integer.
2. *DECx dif* is equivalent to *SUBx S^#1, dif*, but is 1 byte shorter.

DIV Divide

Format

```
opcode divr.rx, quo.mx                2 operand
opcode divr.rx, divd.rx, quo.wx       3 operand
```

Operation

```
quo ← quo / divr;                     !2 operand
quo ← divd / divr;                    !3 operand
```

Condition codes

```
N ← quo LSS 0;
Z ← quo EQL 0;
V ← {integer overflow} OR {divr EQL 0};
C ← 0;
```

Exceptions

```
integer overflow
divide by zero
```

Opcodes

86	DIVB2	Divide Byte 2 Operand
87	DIVB3	Divide Byte 3 Operand
A6	DIVW2	Divide Word 2 Operand
A7	DIVW3	Divide Word 3 Operand
C6	DIVL2	Divide Long 2 Operand
C7	DIVL3	Divide Long 3 Operand

Description

In 2 operand format, the quotient operand is divided by the divisor operand, and the quotient operand is replaced by the result. In 3 operand format, the dividend operand is divided by the divisor operand, and the quotient operand is replaced by the result.

Notes

1. The remainder, if any, is lost.
2. Division is performed such that the remainder (unless it is 0) has the same sign as the dividend; that is, the result is truncated toward 0.
3. Integer overflow occurs if and only if the largest negative integer is divided by −1. On overflow, operands are affected as in note 4 below.
4. If the divisor operand is 0, then in 2 operand format the quotient operand is not affected; in 3 operand format the quotient operand is replaced by the dividend operand.

EDIV Extended Divide

Format

```
opcode divr.rl, divd.rq, quo.wl, rem.wl
```

Operation

```
quo ← divd / divr;
rem ← REM(divd, divr);
```

Condition Codes

```
N ← quo LSS 0;
Z ← quo EQL 0;
V ← {integer overflow} OR {divr EQL 0};
C ← 0;
```

Exceptions

```
integer overflow
divide by zero
```

Opcodes

7B EDIV Extended Divide

Description

The dividend operand is divided by the divisor operand; the quotient operand is replaced by the quotient; and the remainder operand is replaced by the remainder.

Notes

1. The division is performed such that the remainder operand (unless it is 0) has the same sign as the dividend operand.
2. On overflow, the operands are affected as in note 3 below.
3. If the divisor operand is 0, then the quotient operand is replaced by bits ⟨31:0⟩ of the dividend operand; the remainder operand is replaced by 0.

EMUL Extended Multiply

Format

```
opcode mulr.rl, muld.rl, add.rl, prod.wq
```

Operation

```
prod ← {muld * mulr} + SEXT(add);
```

Condition codes

```
N ← prod LSS 0;
Z ← prod EQL 0;
V ← 0;
C ← 0;
```

Exceptions

none

Opcodes

7A	EMUL	Extended Multiply

Description

The multiplicand operand is multiplied by the multiplier operand, giving a double-length result. The addend operand is sign extended to double length and added to the result. The product operand is replaced by the final result.

INC Increment

Format

```
opcode sum.mx
```

Operation

```
sum ← sum + 1;
```

Condition codes

```
N ← sum LSS 0;
Z ← sum EQL 0;
V ← {integer overflow};
C ← {carry from most significant bit};
```

Exceptions

integer overflow

Opcodes

96	INCB	Increment Byte
B6	INCW	Increment Word
D6	INCL	Increment Long

Description

One is added to the sum operand, and the sum operand is replaced by the result.

Notes

1. Arithmetic overflow occurs if the largest positive integer is incremented. On overflow, the sum operand is replaced by the largest negative integer.
2. *INCx sum* is equivalent to *ADDx S^#1, sum*, but is 1 byte shorter.

MCOM Move Complemented

Format

```
opcode src.rx, dst.wx
```

Operation

```
dst ← NOT src;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

92	MCOMB	Move Complemented Byte
B2	MCOMW	Move Complemented Word
D2	MCOML	Move Complemented Long

Description

The destination operand is replaced by the one's complement of the source operand.

MNEG Move Negated

Format

```
opcode src.rx, dst.wx
```

Operation

```
dst ← -src;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {integer overflow};
C ← dst NEQ 0;
```

Exceptions

integer overflow

Opcodes

8E	MNEGB	Move Negated Byte
AE	MNEGW	Move Negated Word
CE	MNEGL	Move Negated Long

Description

The destination operand is replaced by the negative of the source operand.

Notes

1. Integer overflow occurs if the source operand is the largest negative integer (which has no positive counterpart). On overflow, the destination operand is replaced by the source operand.

MOV — Move

Format

```
opcode src.rx, dst.wx
```

Operation

```
dst ← src;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

90	MOVB	Move Byte
B0	MOVW	Move Word
D0	MOVL	Move Long
7D	MOVQ	Move Quad
7DFD	MOVO	Move Octa

Description

The destination operand is replaced by the source operand.

Notes

1. The MOVO instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

MOVZ — Move Zero-Extended

Format

```
opcode src.rx, dst.wy
```

Operation

```
dst ← ZEXT(src);
```

Condition codes

```
N ← 0;
Z ← dst EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

9B	MOVZBW	Move Zero-Extended Byte to Word
9A	MOVZBL	Move Zero-Extended Byte to Long
3C	MOVZWL	Move Zero-Extended Word to Long

Description

For MOVZBW, bits ⟨7:0⟩ of the destination operand are replaced by the source operand; bits ⟨15:8⟩ are replaced by 0. For MOVZBL, bits ⟨7:0⟩ of the destination operand are replaced by the source operand; bits ⟨31:8⟩ are replaced by 0. For MOVZWL, bits ⟨15:0⟩ of the destination operand are replaced by the source operand; bits ⟨31:16⟩ are replaced by 0.

MUL Multiply

Format

```
opcode mulr.rx, prod.mx                2 operand
opcode mulr.rx, muld.rx, prod.wx       3 operand
```

Operation

```
prod ← prod * mulr;                    !2 operand
prod ← muld * mulr;                    !3 operand
```

Condition codes

```
N ← prod LSS 0;
Z ← prod EQL 0;
V ← {integer overflow};
C ← 0;
```

Exceptions

```
integer overflow
```

Opcodes

84	MULB2	Multiply Byte 2 Operand
85	MULB3	Multiply Byte 3 Operand
A4	MULW2	Multiply Word 2 Operand
A5	MULW3	Multiply Word 3 Operand
C4	MULL2	Multiply Long 2 Operand
C5	MULL3	Multiply Long 3 Operand

Description

In 2 operand format, the product operand is multiplied by the multiplier operand, and then the product operand is replaced by the low half of the double-length result. In 3 operand format, the multiplicand operand is multiplied by the multiplier operand, and then the product operand is replaced by the low half of the double-length result.

Notes

1. Integer overflow occurs if the high half of the double-length result is not equal to the sign extension of the low half.

PUSHL Push Long

Format

```
opcode src.rl  {-(SP).wl}
```

Operation

```
-(SP) ← src;
```

Condition codes

```
N ← src LSS 0;
Z ← src EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

DD PUSHL Push Long

Description

The longword source operand is pushed on the stack.

Notes

1. *PUSHL src* is equivalent to *MOVL src, –(SP)*, but is 1 byte shorter.

ROTL Rotate Long

Format

```
opcode cnt.rb, src.rl, dst.wl
```

Operation

```
dst ← src rotated cnt bits;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

9C ROTL Rotate Long

Description

The source operand is rotated logically by the number of bits specified by the count operand, and the destination operand is replaced by the result. The source operand is unaffected. A positive count operand rotates to the left. A negative count operand rotates to the right. A zero count operand replaces the destination operand with the source operand.

SBWC Subtract With Carry

Format

```
opcode sub.rl, dif.ml
```

Operation

```
dif ← dif - sub - C;
```

Condition codes

```
N ← dif LSS 0;
Z ← dif EQL 0;
V ← {integer overflow};
C ← {borrow into most significant bit};
```

Exceptions

```
integer overflow
```

Opcodes

D9 SBWC Subtract With Carry

Description

The subtrahend operand and the contents of PSL⟨C⟩ are subtracted from the difference operand, and the difference operand is replaced by the result.

Notes

1. On overflow, the difference operand is replaced by the low order bits of the true result.
2. The 2 subtractions in the operation are performed simultaneously.

SUB Subtract

Format

```
opcode sub.rx, dif.mx                2 operand
opcode sub.rx, min.rx, dif.wx        3 operand
```

Operation

```
dif ← dif - sub;                     !2 operand
dif ← min - sub;                     !3 operand
```

Condition codes

```
N ← dif LSS 0;
Z ← dif EQL 0;
V ← {integer overflow};
C ← {borrow into most significant bit};
```

Exceptions

integer overflow

Opcodes

82	SUBB2	Subtract Byte 2 Operand
83	SUBB3	Subtract Byte 3 Operand
A2	SUBW2	Subtract Word 2 Operand
A3	SUBW3	Subtract Word 3 Operand
C2	SUBL2	Subtract Long 2 Operand
C3	SUBL3	Subtract Long 3 Operand

Description

In 2 operand format, the subtrahend operand is subtracted from the difference operand, and the difference operand is replaced by the result. In 3 operand format, the subtrahend operand is subtracted from the minuend operand, and the difference operand is replaced by the result.

Notes

1. Integer overflow occurs if the input operands to the subtract are of different signs and the sign of the result is the sign of the subtrahend. On overflow, the difference operand is replaced by the low order bits of the true result.

TST — Test

Format

```
opcode src.rx
```

Operation

```
src - 0;
```

Condition codes

```
N ← src LSS 0;
Z ← src EQL 0;
V ← 0;
C ← 0;
```

Exceptions

none

Opcodes

95	TSTB	Test Byte
B5	TSTW	Test Word
D5	TSTL	Test Long

Description

The condition codes are affected according to the value of the source operand.

Notes

1. *TSTx src* is equivalent to *CMPx src, S^#0*, but is 1 byte shorter.

XOR — Exclusive-OR

Format

```
opcode mask.rx, dst.mx              2 operand
opcode mask.rx, src.rx, dst.wx      3 operand
```

Operation

```
dst ← dst XOR mask;                 !2 operand
dst ← src XOR mask;                 !3 operand
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

8C	XORB2	Exclusive-OR Byte 2 Operand
8D	XORB3	Exclusive-OR Byte 3 Operand
AC	XORW2	Exclusive-OR Word 2 Operand
AD	XORW3	Exclusive-OR Word 3 Operand
CC	XORL2	Exclusive-OR Long 2 Operand
CD	XORL3	Exclusive-OR Long 3 Operand

Description

In 2 operand format, the mask operand is XORed with the destination operand, and then the destination operand is replaced by the result. In 3 operand format, the mask operand is XORed with the source operand, and then the destination operand is replaced by the result.

3.4 Address Instructions

This class of instructions evaluates the address of the source operand and places the effective address in the destination operand. VAX processors are designed to optimize the performance of these instructions, except where otherwise noted.

MOVA Move Address

Format

```
opcode src.ax, dst.wl
```

Operation

```
dst ← src;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

9E	MOVAB	Move Address Byte
3E	MOVAW	Move Address Word
DE	MOVAL	Move Address Long,
	MOVAF	Move Address F_floating
7E	MOVAQ	Move Address Quad,
	MOVAD	Move Address D_floating,
	MOVAG	Move Address G_floating
7EFD	MOVAH	Move Address H_floating,
	MOVAO	Move Address Octa

Description

The destination operand is replaced by the source operand. The context in which the source operand is evaluated is given by the data type of the instruction. The operand whose address replaces the destination operand is not referenced.

Notes

1. The source operand is of address access type, which causes the address of the specified operand to be moved.
2. The MOVAH and MOVAO instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

PUSHA Push Address

Format

```
opcode src.ax  {-(SP).wl}
```

Operation

```
-(SP) ← src;
```

Condition codes

```
N ← src LSS 0;
Z ← src EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

9F	PUSHAB	Push Address Byte
3F	PUSHAW	Push Address Word
DF	PUSHAL	Push Address Long,
	PUSHAF	Push Address F_floating
7F	PUSHAQ	Push Address Quad,
	PUSHAD	Push Address D_floating,
	PUSHAG	Push Address G_floating
7FFD	PUSHAH	Push Address H_floating,
	PUSHAO	Push Address Octa

Description

The source operand is pushed on the stack. The context in which the source operand is evaluated is given by the data type of the instruction. The operand whose address is pushed is not referenced.

Notes

1. *PUSHAx src* is equivalent to *MOVAx src, –(SP)*, but is 1 byte shorter.
2. The source operand is of address access type, which causes the address of the specified operand to be pushed.
3. The PUSHAH and PUSHAO instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

3.5 *Variable-Length Bit-Field Instructions*

A variable-length bit field is specified by three operands:

1. A longword position operand.
2. A byte field-size operand that must be in the range 0 through 32; otherwise, a reserved-operand fault occurs.
3. A base address (relative to which the position is used to locate the bit field). The address is obtained from an operand of address access type. Unlike other instances of operand specifiers of address access type, however, register mode may be designated in the operand specifier. In this case, the field is contained in the register n designated by the operand specifier (or register n+1 concatenated with register n; see Chapter 1). If the field is contained in a register and size is not zero, the position operand must have a value in the range 0 through 31, or a reserved operand fault occurs.

To simplify the description of the bit-field instructions, a macro FIELD(pos, size, address) is introduced with the following expansion (if size NEQ 0):

```
FIELD(pos, size, address)

= (address + SEXT(pos<31:3>))<{size - 1} + pos<2:0>:pos<2:0>>
      !if address not specified by register mode

= {R[n+1]'Rn}<{size - 1} + pos:pos>
      !if address specified by register mode and
      !pos + size GTRU 32

= Rn<{size - 1} + pos:pos>
      !if address specified by register mode and
      !pos + size LEQU 32
```

In general, the number of bytes referenced by the contents { } operator above is:

```
1 + {{{size - 1} + pos<2:0>} / 8}
```

In the case of INSV, however, the contents operator may reference more than this, up to and including the entirety of the aligned longword(s) that contains the field.

Zero bytes are referenced if the field size is 0.

CMP — Compare Field

Format

```
opcode pos.rl, size.rb, base.vb, {field.rv}, src.rl
```

Operation

```
if size NEQU 0 then
   tmp ← SEXT(FIELD(pos, size, base))  !CMPV
   tmp ← ZEXT(FIELD(pos, size, base))  !CMPZV
else
   tmp ← 0;
tmp - src;
```

Condition codes

```
N ← tmp LSS src;
Z ← tmp EQL src;
V ← 0;
C ← tmp LSSU src;
```

Exceptions

```
reserved operand
```

Opcodes

EC	CMPV	Compare Field
ED	CMPZV	Compare Zero-Extended Field

Description

The field specified by the position, size, and base operands is compared with the source operand. For CMPV, the source operand is compared with the sign-extended field. For CMPZV, the source operand is compared with the zero-extended field. The only action is to affect the condition codes.

Notes

1. A reserved operand fault occurs if:
   - size GTRU 32, or
   - pos GTRU 31, and size NEQ 0, and the field is contained in the registers.
2. On a reserved operand fault, the condition codes are UNPREDICTABLE.

EXT — Extract Field

Format

```
opcode pos.rl, size.rb, base.vb, {field.rv}, dst.wl
```

Operation

```
if size NEQU 0 then
   dst ← SEXT(FIELD(pos, size, base))  !EXTV
   dst ← ZEXT(FIELD(pos, size, base))  !EXTZV
else
   dst ← 0;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← 0;
C ← C;
```

Exceptions

```
reserved operand
```

Opcodes

EE	EXTV	Extract Field
EF	EXTZV	Extract Zero-Extended Field

Description

The destination operand is replaced by the field specified by the position, size, and base operands. For EXTV the field is first sign-extended; for EXTZV it is zero-extended. If the size operand is 0, the only actions are to replace the destination operand with 0 and to affect the condition codes.

Notes

1. A reserved operand fault occurs if:
   - size GTRU 32, or
   - pos GTRU 31, and size NEQ 0, and the field is contained in the registers.
2. On a reserved operand fault, the destination operand is unaffected and the condition codes are UNPREDICTABLE.

FF Find First

Format

```
opcode startpos.rl, size.rb, base.vb, {field.rv}, findpos.wl
```

Operation

```
state = if {FFS} then 1 else 0;
if size NEQU 0 then
   begin
   tmp1 ← FIELD(startpos, size, base);
   tmp2 ← 0;
   while {tmp1<tmp2> NEQ state} AND
         {tmp2 LEQU {size - 1}} do
      tmp2 ← tmp2 + 1;
   findpos ← startpos + tmp2;
   end
else
   findpos ← startpos;
```

Condition codes

```
N ← 0;
Z ← {bit not found};
V ← 0;
C ← 0;
```

Exceptions

reserved operand

Opcodes

EB	FFC	Find First Clear
EA	FFS	Find First Set

Description

A field specified by the start position, size, and base operands is extracted. The field is tested for a bit in the state indicated by the instruction, starting at bit 0 and extending to the highest bit in the field. If a bit in the indicated state is found, the find position operand is replaced by the position of the bit, and PSL⟨Z⟩ is cleared. If no bit in the indicated state is found, the find position operand is replaced by the position (relative to the base) of a bit one position to the left of the specified field, and PSL⟨Z⟩ is set. If the size operand is 0, the find position operand is replaced by the start position operand, and PSL⟨Z⟩ is set.

Notes

1. A reserved operand fault occurs if:
   - size GTRU 32, or
   - startpos GTRU 31, and size NEQ 0, and the field is contained in the registers.
2. On a reserved operand fault, the find position operand is unaffected and the condition codes are UNPREDICTABLE.

INSV Insert Field

Format

```
opcode src.rl, pos.rl, size.rb, base.vb, {field.mv}
```

Operation

```
if size NEQU 0 then
   FIELD(pos, size, base) ← src<{size-1}:0>;
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

```
reserved operand
```

Opcodes

F0 INSV Insert Field

Description

The field specified by the position, size, and base operands is replaced by bits ⟨size−1:0⟩ of the source operand. If the size operand is 0, the instruction has no effect.

Notes

1. When executing INSV, a processor may read in the entire aligned longword(s) that contains the field, replace the field portion of the aligned longword(s) with the source operand, and write back the entire aligned longword(s). Because of this, data written to the non-field portion of the aligned longword(s) in memory by another processor or I/O device during the execution of the INSV may be written over when the INSV is completed.
2. A reserved operand fault occurs if:
   - size GTRU 32, or
   - pos GTRU 31, and size NEQ 0, and the field is contained in the registers.
3. On a reserved operand fault, the field is unaffected and the condition codes are UNPREDICTABLE.

3.6 Control Instructions

This class of instructions controls the flow of program execution. VAX processors are designed to optimize the performance of these instructions with one exception. Add Compare and Branch instructions that operate on floating-point operands (ACBF, ACBD, ACBG, and ACBH) belong to an instruction group that is optional to implement; refer to Chapter 11.

In most implementations of the VAX architecture, improved execution speed will result if the target of a control instruction is on an aligned longword boundary.

ACB — Add Compare and Branch

Format

```
opcode limit.rx, add.rx, index.mx, displ.bw
```

Operation

```
index ← index + add;
if {{add GEQ 0} AND {index LEQ limit}} OR
   {{add LSS 0} AND {index GEQ limit}} then
   PC ← PC + SEXT(displ);
```

Condition codes

```
N ← index LSS 0;
Z ← index EQL 0;
V ← {integer overflow};
C ← C;
```

Exceptions

```
integer overflow
floating overflow
floating underflow
reserved operand
```

Opcodes

9D	ACBB	Add Compare and Branch Byte
3D	ACBW	Add Compare and Branch Word
F1	ACBL	Add Compare and Branch Long
4F	ACBF	Add Compare and Branch F_floating
6F	ACBD	Add Compare and Branch D_floating
4FFD	ACBG	Add Compare and Branch G_floating
6FFD	ACBH	Add Compare and Branch H_floating

Description

The addend operand is added to the index operand, and the index operand is replaced by the result. The index operand is compared with the limit operand. If the addend operand is positive (or 0) and the comparison is less than or equal, or if the addend is negative and the comparison is greater than or equal, the sign-extended

branch displacement is added to PC. PC is then replaced by the result.

1. ACB directly implements the general FOR or DO loops in high-level languages since the sense of the comparison between index and limit depends on the sign of the addend.
2. On integer overflow, the index operand is replaced by the low order bits of the true result. Comparison and branch determination proceed normally on the updated index operand.
3. On floating underflow, if PSL⟨FU⟩ is clear, the index operand is replaced by 0 and comparison and branch determination proceed normally. A fault occurs if PSL⟨FU⟩ is set and the index operand is unaffected.
4. On floating overflow, the instruction takes a floating overflow fault and the index operand is unaffected.
5. On a reserved operand fault, the index operand is unaffected and the condition codes are UNPREDICTABLE.
6. The ACBx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11.
7. The ACBx instructions are in the emulated-only group and are unlikely to be implemented in future VAX processors. Software developers are advised to avoid using these instructions.

AOBLEQ Add One and Branch Less Than or Equal

Format

```
opcode limit.rl, index.ml, displ.bb
```

Operation

```
index ← index + 1;
if index LEQ limit then
   PC ← PC + SEXT(displ);
```

Condition codes

```
N ← index LSS 0;
Z ← index EQL 0;
V ← {integer overflow};
C ← C;
```

Exceptions

```
integer overflow
```

Opcodes

F3 AOBLEQ Add One and Branch Less Than or Equal

Description

One is added to the index operand, and the index operand is replaced by the result. The index operand is compared with the limit operand. If it is less than or equal, the sign-extended branch displacement is added to PC. PC is then replaced by the result.

Notes

1. Integer overflow occurs if the index operand before addition is the largest positive integer. On overflow, the index operand is replaced by the largest negative integer and the branch is taken.
2. The C-bit is unaffected.

AOBLSS Add One and Branch Less Than

Format

```
opcode limit.rl, index.ml, displ.bb
```

Operation

```
index ← index + 1;
if index LSS limit then
   PC ← PC + SEXT(displ);
```

Condition codes

```
N ← index LSS 0;
Z ← index EQL 0;
V ← {integer overflow};
C ← C;
```

Exceptions

```
integer overflow
```

Opcodes

F2 AOBLSS Add One and Branch Less Than

Description

One is added to the index operand, and the index operand is replaced by the result. The index operand is compared with the limit operand. If it is less than, the sign-extended branch displacement is added to the PC and PC is replaced by the result.

Notes

1. Integer overflow occurs if the index operand before addition is the largest positive integer. On overflow, the index operand is replaced by the largest negative integer; thus, the branch is taken (unless the limit operand is the largest negative integer).
2. The C-bit is unaffected.

B Branch on (condition)

Format

```
opcode displ.bb
```

Operation

```
if condition then PC ← PC + SEXT(displ);
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

none

Opcodes

14	BGTR	(N OR Z) EQL 0	Branch on Greater Than signed
15	BLEQ	(N OR Z) EQL 1	Branch on Less Than or Equal signed
12	BNEQ	Z EQL 0	Branch on Not Equal signed
	BNEQU		Branch on Not Equal Unsigned
13	BEQL	Z EQL 1	Branch on Equal signed
	BEQLU		Branch on Equal Unsigned
18	BGEQ	N EQL 0	Branch on Greater Than or Equal signed
19	BLSS	N EQL 1	Branch on Less Than signed
1A	BGTRU	(C OR Z) EQL 0	Branch on Greater Than Unsigned
1B	BLEQU	(C OR Z) EQL 1	Branch Less Than or Equal Unsigned
1C	BVC	V EQL 0	Branch on Overflow Clear
1D	BVS	V EQL 1	Branch on Overflow Set
1E	BGEQU	C EQL 0	Branch on Greater Than or Equal Unsigned
	BCC		Branch on Carry Clear
1F	BLSSU	C EQL 1	Branch on Less Than Unsigned
	BCS		Branch on Carry Set

Description

The condition codes are tested and, if the condition indicated by the instruction is met, the sign-extended branch displacement is added to the PC. PC is then replaced by the result.

Notes

1. The VAX conditional branch instructions permit considerable flexibility in branching but require care in choosing the correct branch instruction. The conditional branch instructions are best seen as three overlapping groups:

   a. Overflow and Carry Group

```
BVS   V EQL 1
BVC   V EQL 0
BCS   C EQL 1
BCC   C EQL 0
```

These instructions are typically used to check for overflow (when overflow traps are not enabled), for multiprecision arithmetic, and for other special purposes.

b. Unsigned Group

```
BLSSU   C EQL 1
BLEQU   {C OR Z} EQL 1
BEQLU   Z EQL 1
BNEQU   Z EQL 0
BGEQU   C EQL 0
BGTRU   {C OR Z} EQL 0
```

These instructions typically follow integer and field instructions where the operands are treated as unsigned integers, address instructions, and character-string instructions.

c. Signed Group

```
BLSS    N EQL 1
BLEQ    {N OR Z} EQL 1
BEQL    Z EQL 1
BNEQ    Z EQL 0
BGEQ    N EQL 0
BGTR    {N OR Z} EQL 0
```

These instructions typically follow integer and field instructions where the operands are treated as signed integers, floating-point instructions, and decimal-string instructions.

BB Branch on Bit

Format

```
opcode pos.rl, base.vb, displ.bb, {field.rv}
```

Operation

```
teststate = if {BBS} then 1 else 0;
if FIELD(pos, 1, base) EQL teststate then
   PC ← PC + SEXT(displ);
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

```
reserved operand
```

Opcodes

E0	BBS	Branch on Bit Set
E1	BBC	Branch on Bit Clear

Description

The single-bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to PC and PC is replaced by the result.

Notes

1. See section 3.5 earlier in this chapter for a definition of FIELD.
2. A reserved operand fault occurs if pos GTRU 31 and the bit is contained in a register.
3. On a reserved operand fault, the condition codes are UNPREDICTABLE.

BB — Branch on Bit (and modify without interlock)

Format

```
opcode pos.rl, base.vb, displ.bb, {field.mv}
```

Operation

```
teststate = if {BBSS or BBSC} then 1 else 0;
newstate = if {BBSS or BBCS} then 1 else 0;
tmp ← FIELD(pos, 1, base);
FIELD(pos, 1, base) ← newstate;
if tmp EQL teststate then
   PC ← PC + SEXT(displ);
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

reserved operand

Opcodes

E2	BBSS	Branch on Bit Set and Set
E3	BBCS	Branch on Bit Clear and Set
E4	BBSC	Branch on Bit Set and Clear
E5	BBCC	Branch on Bit Clear and Clear

Description

The single-bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to PC and PC is replaced by the result. Regardless of whether the branch is taken or not, the tested bit is put in the new state as indicated by the instruction.

Notes

1. See section 3.5 earlier in this chapter for a definition of FIELD.
2. A reserved operand fault occurs if pos GTRU 31 and the bit is contained in a register.
3. On a reserved operand fault, the field is unaffected and the condition codes are UNPREDICTABLE.
4. The modification of the bit is not an interlocked operation. See BBSSI and BBCCI for interlocking instructions.

BB Branch on Bit Interlocked

Format

```
opcode pos.rl, base.vb, displ.bb, {field.mv}
```

Operation

```
teststate = if {BBSSI} then 1 else 0;
newstate = teststate;
{set interlock};
tmp ← FIELD(pos, 1, base);
FIELD(pos, 1, base) ← newstate;
{release interlock};
if tmp EQL teststate then
   PC ← PC + SEXT(displ);
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

reserved operand

Opcodes

E6	BBSSI	Branch on Bit Set and Set Interlocked
E7	BBCCI	Branch on Bit Clear and Clear Interlocked

Description

The single-bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to PC and PC is replaced by the result. Regardless of whether the branch is effected or not, the tested bit is put in the new state as indicated by the instruction. If the bit is contained in memory, the reading of the state of the bit and the setting of it to the new state is an interlocked operation. No other processor or I/O device can do an interlocked access on the bit during the interlocked operation.

Notes

1. See section 3.5 earlier in this chapter for a definition of FIELD.
2. A reserved operand fault occurs if pos GTRU 31 and the bit is contained in a register.
3. On a reserved operand fault, the field is unaffected and the condition codes are UNPREDICTABLE.
4. Except for memory interlocking, BBSSI is equivalent to BBSS and BBCCI is equivalent to BBCC.
5. This instruction is designed to modify interlocks with other processors or devices. For example, to implement a "busy waiting" protocol:

```
1$:     BBSSI bit, base, 2$ ; Try to obtain interlock.
                            ; Success: fall through.
                            ; Failure: branch to 2$.
```

```
; Insert code sequence that requires the interlock here.

        BRB     3$          ; Branch around busy loop.

; Wait for the busy interlock to become free, using a
; non-interlocked instruction.  When the lock becomes
; free, try the interlocked instruction again.

2$:     BBS     bit, base, 2$ ; Loop until lock is free.
        BRB     1$            ; Retry operation.

3$:     BBCCI   bit, base, 4$   ; Release the interlock.
4$:
```

BLB Branch on Low Bit

Format

```
opcode src.rl, displ.bb
```

Operation

```
teststate = if {BLBS} then 1 else 0;
if src<0> EQL teststate then
   PC ← PC + SEXT(displ);
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

none

Opcodes

E8	BLBS	Branch on Low Bit Set
E9	BLBC	Branch on Low Bit Clear

Description

The low bit (bit 0) of the source operand is tested; if it is equal to the test state indicated by the instruction, the sign-extended branch displacement is added to PC. PC is then replaced by the result.

BR Branch

Format

```
opcode displ.bx
```

Operation

```
PC ← PC + SEXT(displ);
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

none

Opcodes

11	BRB	Branch With Byte Displacement
31	BRW	Branch With Word Displacement

Description

The sign-extended branch displacement is added to PC, and PC is replaced by the result.

BSB — Branch to Subroutine

Format

```
opcode displ.bx, {-(SP).wl}
```

Operation

```
-(SP) ← PC;
PC ← PC + SEXT(displ);
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

none

Opcodes

10	BSBB	Branch to Subroutine With Byte Displacement
30	BSBW	Branch to Subroutine With Word Displacement

Description

PC is pushed on the stack as a longword. The sign-extended branch displacement is added to PC, and PC is replaced by the result.

CASE Case

Format

```
opcode selector.rx, base.rx, limit.rx,
       displ[0].bw,..., displ[limit].bw
```

Operation

```
tmp ← selector - base;
if tmp LEQU limit then
   tmp1 ← SEXT(displ[tmp])
else
   tmp1 ← {2 + 2 * ZEXT(limit)};
PC ← PC + tmp1
```

Condition codes

```
N ← tmp LSS limit;
Z ← tmp EQL limit;
V ← 0;
C ← tmp LSSU limit;
```

Exceptions

none

Opcodes

8F	CASEB	Case Byte
AF	CASEW	Case Word
CF	CASEL	Case Long

Description

The base operand is subtracted from the selector operand, and a temporary is replaced by the result. The temporary is compared with the limit operand; if it is less than or equal unsigned, a branch displacement selected by the temporary value is added to PC. PC is then replaced by the result. Otherwise, two times the sum of the limit operand and 1 is added to PC, and PC is replaced by the result. This causes PC to be moved past the array of branch displacements. Regardless of the branch taken, the condition codes are affected by the comparison of the temporary operand with the limit operand.

Notes

1. After operand evaluation, PC is pointing at displ[0], not at the next instruction. The branch displacements are relative to the address of displ[0].
2. The selector and base operands can both be considered either as signed or unsigned integers.
3. The Pascal statement:

```
case i of
32:         x := sin(x);
33:         x := cos(x);
34:         x := exp(x);
35:         x := ln(x);
36, 37:     x := arctanh(x);
```

```
otherwise   x := reserved
end
```

is translated by the VAX Pascal compiler to:

```
      casel   i, #32, #<37-32>
 1$:  .word   sin - 1$        ; Selector is 32.
      .word   cos - 1$        ; Selector is 33.
      .word   exp - 1$        ; Selector is 34.
      .word   ln - 1$         ; Selector is 35.
      .word   arctanh - 1$    ; Selector is 36.
      .word   arctanh - 1$    ; Selector is 37.
otherwise:
      movl    reserved, x     ; Selector is less than
                              ; 32 or greater than 37.
```

JMP — Jump

Format

```
opcode dst.ab
```

Operation

```
PC ← dst;
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

none

Opcodes

17 JMP Jump

Description

PC is replaced by the destination operand.

Notes

1. If the destination operand's addressing mode is immediate mode, the results of the instruction are UNPREDICTABLE.

JSB — Jump to Subroutine

Format

```
opcode dst.ab, {-(SP).wl}
```

Operation

```
-(SP) ← PC;
PC ← dst;
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

none

Opcodes

16 JSB Jump to Subroutine

Description

PC is pushed on the stack as a longword. PC is replaced by the destination operand.

Notes

1. Since the operand specifier conventions cause the evaluation of the destination operand before saving PC, JSB can be used for coroutine calls with the stack used for linkage. The form of such a call is *JSB @(SP)+*.
2. If the destination operand's addressing mode is immediate mode, the results of the instruction are UNPREDICTABLE.

RSB — Return from Subroutine

Format

```
opcode {(SP)+.rl}
```

Operation

```
PC ← (SP)+;
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

none

Opcodes

05 RSB Return From Subroutine

Description

PC is replaced by a longword popped from the stack.

Notes

1. RSB is used to return from subroutines called by the BSBB, BSBW, and JSB instructions.
2. *RSB* is equivalent to *JMP @(SP)+*, but is 1 byte shorter.

SOBGEQ — Subtract One and Branch Greater Than or Equal

Format

```
opcode index.ml, displ.bb
```

Operation

```
index ← index - 1;
if index GEQ 0 then
   PC ← PC + SEXT(displ);
```

Condition codes

```
N ← index LSS 0;
Z ← index EQL 0;
V ← {integer overflow};
C ← C;
```

Exceptions

integer overflow

Opcodes

F4 SOBGEQ Subtract One and Branch Greater Than or Equal

Description

One is subtracted from the index operand, and the index operand is replaced by the result. If the index operand is greater than or equal to 0, the sign-extended branch displacement is added to PC. PC is then replaced by the result.

Notes

1. Integer overflow occurs if the index operand before subtraction is the largest negative integer. On overflow, the index operand is replaced by the largest positive integer; thus, the branch is taken.
2. The C-bit is unaffected.

SOBGTR Subtract One and Branch Greater Than

Format

```
opcode index.ml, displ.bb
```

Operation

```
index ← index - 1;
if index GTR 0 then
   PC ← PC + SEXT(displ);
```

Condition codes

```
N ← index LSS 0;
Z ← index EQL 0;
V ← {integer overflow};
C ← C;
```

Exceptions

```
integer overflow
```

Opcodes

F5 SOBGTR Subtract One and Branch Greater Than

Description

One is subtracted from the index operand, and the index operand is replaced by the result. If the index operand is greater than 0, the sign-extended branch displacement is added to PC. PC is then replaced by the result.

Notes

1. Integer overflow occurs if the index operand before subtraction is the largest negative integer. On overflow, the index operand is replaced by the largest positive integer; thus, the branch is taken.
2. The C-bit is unaffected.

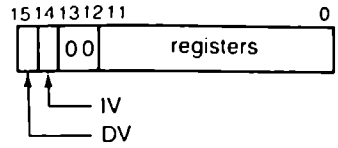


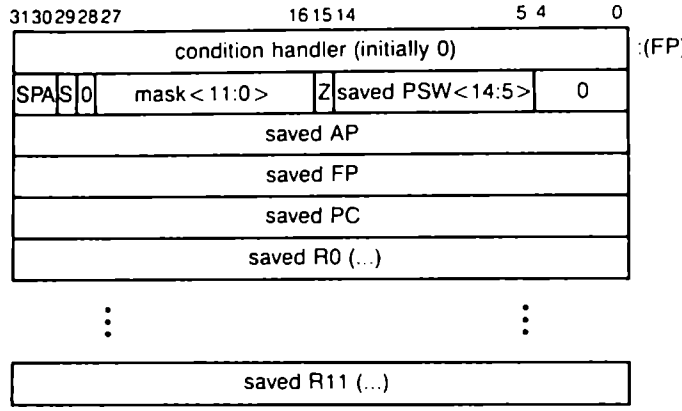
Figure 3.1
Procedure Entry Mask

3.7 *Procedure-Call Instructions*

Three instructions are used to implement a standard procedure-calling interface. (Refer to *VAX/VMS Introduction to System Routines* for the procedure-calling standard.) Two instructions implement the call to the procedure; the third implements the matching return. The CALLG instruction calls a procedure with the argument list in an arbitrary location. The CALLS instruction calls a procedure with the argument list on the stack. Upon return after a CALLS, this list is automatically removed from the stack. Both call instructions specify the address of the entry point of the procedure being called. The entry point is assumed to consist of a word termed the "entry mask" followed by the procedure's instructions. The procedure terminates by executing a RET instruction.

The entry mask specifies the subprocedure's register use and overflow enables, as shown in Figure 3.1. On CALL, the stack is aligned to a longword boundary and the trap enables in the PSW are set to a known state to ensure consistent behavior of the called procedure. Integer overflow-enable and decimal overflow-enable are affected according to bits ⟨14⟩ and ⟨15⟩ of the entry mask, respectively. Floating underflow-enable is cleared. The registers R11 through R0 specified by bits ⟨11⟩ through ⟨0⟩, respectively, are saved on the stack and are restored by the RET instruction. In addition, PC, SP, FP, and AP are always preserved by the CALL instructions and restored by the RET instruction.

All external procedure calls generated by standard DIGITAL language processors and all intermodule calls to major VAX software subsystems comply with the procedure-calling software standard. The procedure-calling standard requires that all registers in the range R2 through R11 used in the procedure must appear in the



(0 to 3 bytes specified by SPA, Stack Pointer Alignment)

S = Set if CALLS; clear if CALLG.
Z = Always cleared by CALL. Can be set by software to force a reserved operand fault on a RET.

Figure 3.2
Procedure-Call Stack Frame

mask. R0 and R1 are not preserved by any called procedure that complies with the procedure-calling standard.

To preserve the state, the CALL instructions form a structure on the stack termed a call frame, or stack frame, shown in Figure 3.2. This structure contains the saved registers, the saved PSW, the register save mask, and several control bits. The frame also includes a longword that the CALL instructions clear; this is used to implement the VAX/VMS condition-handling facility. (Refer to the VAX/VMS document set.) At the end of execution of the CALL instruction, FP contains the address of the stack frame. The RET instruction uses the contents of FP to find the stack frame and restore state. The condition-handling facility assumes that FP always points to the stack frame. Note that the saved condition codes and the saved trace enable (PSW⟨T⟩) are cleared.

The contents of the frame PSW⟨3:0⟩ at the time RET is executed will become the condition codes resulting from the execution of the procedure. Similarly, the content of the frame PSW⟨4⟩ at the time the RET is executed will become the PSW⟨T⟩ bit.

CALLG Call Procedure with General Argument List

Format

```
opcode arglist.ab, dst.ab, {-(SP).w*}
```

Operation

```
tmp2 ← (dst);
tmp1 ← SP;
SP<1:0> ← 0;
for tmp3 ← 11 step -1 until 0 do
   if tmp2<tmp3> EQL 1 then
      -(SP) ← R[tmp3];
-(SP) ← PC;
-(SP) ← FP;
-(SP) ← AP;
PSW<N,Z,V,C> ← 0;
-(SP) ← tmp1<1:0>'0'0'tmp2<11:0>'0'PSW<14:5>'0<4:0>;
-(SP) ← 0;
FP ← SP;
AP ← arglist;
PSW<DV> ← tmp2<15>;
PSW<IV> ← tmp2<14>;
PSW<FU> ← 0;
PC ← dst + 2;
```

Condition codes

```
N ← 0;
Z ← 0;
V ← 0;
C ← 0;
```

Exceptions

```
reserved operand
```

Opcodes

FA CALLG Call Procedure with General Argument List

Description

SP is saved in a temporary, and then bits ⟨1:0⟩ are replaced by 0 so that the stack is longword aligned. The procedure entry mask is scanned from bits ⟨11⟩ to ⟨0⟩, and the contents of registers whose number corresponds to set bits in the mask are pushed on the stack as longwords. PC, FP, and AP are pushed on the stack as longwords. The condition codes are cleared. A longword containing the following is pushed on the stack: the saved two low bits of SP in bits ⟨31:30⟩, a 0 in bit ⟨29⟩ and bit ⟨28⟩, the low 12 bits of the procedure entry mask in bits ⟨27:16⟩, a 0 in bit ⟨15⟩, and PSW⟨14:0⟩ in bits ⟨14:0⟩ with T and the condition codes cleared. A longword 0 is pushed on the stack. FP is replaced by SP. AP is replaced by the arglist operand. The trap-enables in the PSW are set to a known state. Integer overflow and decimal overflow are affected according to bits ⟨14⟩ and ⟨15⟩, respectively, of the entry mask; floating underflow is cleared. The T-bit is unaffected. PC is

replaced by the sum of destination operand plus 2, which transfers control to the called procedure at the byte beyond the entry mask.

Notes

1. If bits ⟨13:12⟩ of the entry mask are not 0, a reserved operand fault occurs.
2. On a reserved operand fault, condition codes are UNPREDICTABLE.
3. The procedure-calling standard and the condition-handling facility require the following register-saving conventions. R0 and R1 are always available for function return values and are never saved in the entry mask. All registers R2 through R11 that are modified in the called procedure must be preserved in the mask.
4. The alignment bytes left on the stack are UNPREDICTABLE. They may, for example, be written with zeros when the stack is aligned.
5. If the destination operand's addressing mode is immediate mode, the results of the instruction are UNPREDICTABLE.

CALLS — Call Procedure with Stack Argument List

Format

```
opcode numarg.rl, dst.ab, {-(SP).w*}
```

Operation

```
tmp2 ← (dst);
-(SP) ← numarg;
tmp1 ← SP;
SP<1:0> ← 0;

for tmp3 ← 11 step -1 until 0 do
   if tmp2<tmp3> EQL 1 then
      -(SP) ← R[tmp3];
-(SP) ← PC;
-(SP) ← FP;
-(SP) ← AP;
PSW<N,Z,V,C> ← 0;
-(SP) ← tmp1<1:0>'1'0'tmp2<11:0>'0'PSW<14:5>'0<4:0>;
-(SP) ← 0;
FP ← SP;
AP ← tmp1;
PSW<DV> ← tmp2<15>;
PSW<IV> ← tmp2<14>;
PSW<FU> ← 0;
PC ← dst + 2;
```

Condition codes

```
N ← 0;
Z ← 0;
V ← 0;
C ← 0;
```

Exceptions

reserved operand

Opcodes

FB CALLS Call Procedure with Stack Argument List

Description

The numarg operand is pushed on the stack as a longword. (Byte 0 contains the number of arguments; high-order 24 bits are used by DIGITAL software.) SP is saved in a temporary, and then bits ⟨1:0⟩ of SP are replaced by 0 so that the stack is longword aligned. The procedure entry mask is scanned from bit ⟨11⟩ to bit ⟨0⟩, and the contents of registers whose number corresponds to set bits in the mask are pushed on the stack. PC, FP, and AP are pushed on the stack as longwords. The condition codes are cleared. A longword containing the following is pushed on the stack: saved two low bits of SP in bits ⟨31:30⟩, a 1 in bit ⟨29⟩, a 0 in bit ⟨28⟩, the low 12 bits of the procedure entry mask in bits ⟨27:16⟩, a 0 in bit ⟨15⟩, and PSW⟨14:0⟩ in bits ⟨14:0⟩ with T and the condition codes cleared. A longword 0 is pushed on the stack. FP is replaced by SP. AP

CALLS

is set to the value of the stack pointer after the numarg operand was pushed on the stack. The trap-enables in the PSW are set to a known state. Integer overflow and decimal overflow are affected according to bits ⟨14⟩ and ⟨15⟩, respectively, of the entry mask; floating underflow is cleared. T-bit is unaffected. PC is replaced by the sum of destination operand plus 2, which transfers control to the called procedure at the byte beyond the entry mask.

Notes

1. If bits ⟨13:12⟩ of the entry mask are not 0, a reserved operand fault occurs.
2. On a reserved operand fault, the condition codes are UNPREDICTABLE.
3. Normal use is to push the arglist onto the stack in reverse order prior to the CALLS. On return, the arglist is removed from the stack automatically.
4. The procedure-calling standard and the condition-handling facility require the following register-saving conventions. R0 and R1 are always available for function return values and are never saved in the entry mask. All registers R2 through R11 that are modified in the called procedure must be preserved in the entry mask.
5. The alignment bytes left on the stack are UNPREDICTABLE. They may, for example, be written with zeros when the stack is aligned.
6. If the destination operand's addressing mode is immediate mode, the results of the instruction are UNPREDICTABLE.
7. In order to facilitate efficient implementations of the CALLS instruction, a restriction is placed on addressing mode combinations involving logically inconsistent simultaneous use of a value as both a numeric value and an address. Specifically, if the contents of register Rn are used as the numarg operand, and the dst specifier uses the contents of Rn as an address in an addressing mode that modifies Rn (autodecrement, autoincrement, or autoincrement deferred modes), the result is UNPREDICTABLE. For the same reason, if the content of SP (which is implicitly modified) is used as the numarg operand, the result is UNPREDICTABLE.

RET — Return from Procedure

Format

```
opcode {(SP)+.r*}
```

Operation

```
SP ← FP + 4;
tmp1 ← (SP)+;
AP ← (SP)+;
FP ← (SP)+;
PC ← (SP)+;
tmp2 ← tmp1<27:16>;
for tmp3 ← 0 step 1 until 11 do
   if tmp2<tmp3> EQL 1 then
      R[tmp3] ← (SP)+;

SP ← SP + ZEXT{tmp1<31:30>}

if tmp1<29> EQL 1 then
   begin
   tmp4 ← 4 * ZEXT({(SP)+}<7:0>);
   SP ← SP + tmp4;
   end;

PSW ← tmp1<15:0>;
```

Condition codes

```
N ← tmp1<3>;
Z ← tmp1<2>;
V ← tmp1<1>;
C ← tmp1<0>;
```

Exceptions

reserved operand

Opcodes

04 RET Return from Procedure

Description

SP is replaced by FP plus 4. A longword containing the following is popped from the stack and saved in a temporary: stack alignment bits in bits ⟨31:30⟩, a flag distinguishing CALLS from CALLG in bit ⟨29⟩, the low 12 bits of the procedure entry mask in bits ⟨27:16⟩, and a saved PSW in bits ⟨15:0⟩. PC, FP, and AP are replaced by longwords popped from the stack. A register restore mask is formed from bits ⟨27:16⟩ of the temporary. Scanning from bit ⟨0⟩ to bit ⟨11⟩ of the restore mask, the contents of registers whose numbers are indicated by set bits in the mask are replaced by longwords popped from the stack. SP is incremented by ⟨31:30⟩ of the temporary. PSW is replaced by bits ⟨15:0⟩ of the temporary. If bit ⟨29⟩ in the temporary is 1 (indicating that the procedure was called by CALLS), a longword containing the number of arguments is popped from the stack. Four times the unsigned

value of the low byte of this longword is added to SP, and SP is replaced by the result.

Notes

1. A reserved operand fault occurs if tmp1⟨15:8⟩ NEQ 0.
2. On a reserved operand fault, the condition codes are UNPREDICTABLE.
3. The value of tmp1⟨28⟩ is ignored.
4. The procedure-calling standard and condition-handling facility assume that procedures returning a function value or a status code do so in R0 or R0 and R1.
5. If FP⟨1:0⟩ is not zero, or if the stack frame is ill-formed, the results are UNPREDICTABLE.

3.8 *Miscellaneous Instructions*

This class of instructions performs miscellaneous actions such as accessing the PSL, causing breakpoint traps, halting the processor, and performing extended function calls. VAX processors are designed to efficiently implement all these instructions.

BICPSW Bit Clear PSW

Format

```
opcode mask.rw
```

Operation

```
PSW ← PSW AND {NOT mask};
```

Condition codes

```
N ← N AND {NOT mask<3>};
Z ← Z AND {NOT mask<2>};
V ← V AND {NOT mask<1>};
C ← C AND {NOT mask<0>};
```

Exceptions

```
reserved operand
```

Opcodes

B9 BICPSW Bit Clear PSW

Description

PSW is ANDed with the one's complement of the mask operand, and PSW is replaced by the result.

Notes

1. A reserved operand fault occurs if mask ⟨15:8⟩ is not zero. On a reserved operand fault, the PSW is not affected.

BISPSW Bit Set PSW

Format

```
opcode mask.rw
```

Operation

```
PSW ← PSW OR mask;
```

Condition codes

```
N ← N OR mask<3>;
Z ← Z OR mask<2>;
V ← V OR mask<1>;
C ← C OR mask<0>;
```

Exceptions

```
reserved operand
```

Opcodes

B8 BISPSW Bit Set PSW

Description

PSW is ORed with the mask operand, and PSW is replaced by the result.

Notes

1. A reserved operand fault occurs if mask⟨15:8⟩ is not zero. On a reserved operand fault, the PSW is not affected.

BPT — Breakpoint

Format

```
opcode  {-(KSP).w*}
```

Operation

```
{initiate breakpoint fault};
```

Condition codes

```
N ← 0;          !condition codes cleared after BPT fault
Z ← 0;
V ← 0;
C ← 0;
```

Exceptions

none

Opcodes

03 BPT Breakpoint

Description

To understand the operation of this instruction, read Chapter 5. This instruction is used, together with PSL⟨T⟩, to implement debugging facilities.

Notes

1. Because the breakpoint exception is a fault, the PC and PSL pushed on the exception stack reflect the state of the processor at the beginning of the instruction, with PSL⟨TP⟩ clear.

BUG Bugcheck

Format

```
opcode  {message.bx}
```

Operation

```
{fault to report error}
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

```
reserved instruction
```

Opcodes

FEFF	BUGW	Bugcheck with Word Message Identifier
FDFF	BUGL	Bugcheck with Longword Message Identifier

Description

The hardware treats these opcodes as reserved to DIGITAL and faults. The VAX/VMS operating system treats these as requests to report software detected errors. The in-line message identifier is zero extended to a longword (BUGW) and interpreted as a condition value. If the process is privileged to report bugs, a log entry is made. If the process is not privileged, a reserved instruction exception is signaled and is handled by the standard condition handler mechanism.

HALT — Halt

Format

```
opcode {-(KSP).w*}
```

Operation

```
if PSL<CUR_MOD> NEQU 0 then
   {privileged instruction fault}
else
   {halt the processor};
```

Condition codes

```
N ← 0; !If privileged instruction fault
Z ← 0; !condition codes are cleared after
V ← 0; !the fault.  PSL saved on stack
C ← 0; !contains condition codes prior to HALT.

N ← N; !If processor halt
Z ← Z;
V ← V;
C ← C;
```

Exceptions

```
privileged instruction
```

Opcodes

00 HALT Halt

Description

To understand the operation of this instruction it is necessary to read Chapter 5. If the process is running in kernel mode, the processor is halted. Otherwise, a privileged instruction fault occurs.

Notes

1. This opcode is 0 to trap many branches to data.

INDEX Compute Index

Format

```
opcode subscript.rl, low.rl, high.rl,
       size.rl, indexin.rl, indexout.wl
```

Operation

```
indexout ← {indexin + subscript} *size;
if {subscript LSS low} or {subscript GTR high}
then {subscript range trap};
```

Condition codes

```
N ← indexout LSS 0;
Z ← indexout EQL 0;
V ← 0;
C ← 0;
```

Exceptions

subscript range

Opcodes

0A INDEX Compute Index

Description

The indexin operand is added to the subscript operand, and the sum is multiplied by the size operand. The indexout operand is replaced by the result. If the subscript operand is less than the low operand or greater than the high operand, a subscript range trap is taken.

Notes

1. No arithmetic exception other than subscript range can result from this instruction. Thus, no indication is given if overflow occurs in either the add or the multiply step. If overflow occurs on the add step, the sum is the low order 32 bits of the true result. If overflow occurs on the multiply step, the indexout operand is replaced by the low order 32 bits of the true product of the sum and the subscript operand. In the normal use of this instruction, overflow cannot occur without a subscript range trap occurring.
2. The index instruction is useful in index calculations for arrays of the fixed-length data types (integer and floating) and for index calculations for arrays of bit fields, character strings, and decimal strings. The indexin operand permits cascading INDEX instructions for multi-dimensional arrays. For one-dimensional bit field arrays, it also permits introduction of the constant portion of an index calculation, which is not readily absorbed by address arithmetic. The following notes show some of the uses of INDEX.

3. The COBOL statements:

```
01  A-ARRAY.
    02  A PIC X(25) OCCURS 15 TIMES INDEXED BY I.
01  B PIC X(25).

MOVE A(I) TO B.
```

can be translated by a VAX COBOL compiler to:

```
INDEX   I(R11), #^X01, #^X0F, #^X19, #^X00, R0
MOVC3   #^X19, A-25(R11)[R0], B(R11)
```

4. The FORTRAN statements:

```
INTEGER*4       A(11:24), I
A(I) = 1
```

can be translated by a VAX FORTRAN compiler to:

```
INDEX   I(R11), #11, #24, #1, #0, R0
MOVL    #1, A-44(R11)[R0]
```

5. The Pascal statements:

```
var
i : integer;
a : array[11..24] of integer;
  .
  .
  .
a[i] := 1;
```

can be translated by a VAX Pascal compiler to:

```
INDEX   I,#11,#24,#1,#0,R0
MOVZBL  #1,A-44[R0]
```

MOVPSL Move from PSL

Format `opcode dst.wl`

Operation `dst ← PSL`

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions none

Opcodes DC MOVPSL Move from PSL

Description The destination operand is replaced by PSL.

NOP No Operation

Format `opcode`

Operation none

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions none

Opcodes 01 NOP No Operation

Description No operation is performed.

POPR Pop Registers

Format

```
opcode mask.rw, {(SP)+.r*}
```

Operation

```
tmp1 ← mask
for tmp2 ← 0 step 1 until 14 do
   if tmp1<tmp2> EQL 1 then R[tmp2] ← (SP)+;
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

none

Opcodes

BA POPR Pop Registers

Description

The contents of registers whose numbers correspond to set bits in the mask operand are replaced by longwords popped from the stack. R[n] is replaced if mask⟨n⟩ is set. The mask is scanned from bit ⟨0⟩ to bit ⟨14⟩. Bit ⟨15⟩ is ignored.

PUSHR Push Registers

Format

```
opcode mask.rw, {-(SP).w*}
```

Operation

```
tmp1 ← mask;
for tmp2 ← 14 step -1 until 0 do
   if tmp1<tmp2> EQL 1 then -(SP) ← R[tmp2];
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

none

Opcodes

BB PUSHR Push Registers

Description

The contents of registers whose numbers correspond to set bits in the mask operand are pushed on the stack as longwords. R[n] is pushed if mask⟨n⟩ is set. The mask is scanned from bit ⟨14⟩ to bit ⟨0⟩. Bit ⟨15⟩ is ignored.

Notes

1. The order of pushing is specified so that the contents of higher numbered registers are stored at higher memory addresses. This results in, for example, a quadword datum stored in adjacent registers being stored by PUSHR in memory in the correct order.

XFC Extended Function Call

Format

```
opcode  {unspecified operands}
```

Operation

```
{XFC fault};
```

Condition codes

```
N ← 0;
Z ← 0;
V ← 0;
C ← 0;
```

Exceptions

none

Opcodes

FC XFC Extended Function Call

Description

To understand the operation of this instruction, read Chapter 5. This instruction provides for user-defined extensions to the instruction set.

3.9 Queue Instructions

A queue is a circular, doubly linked list. A queue entry is specified by its address. The VAX architecture supports two distinct types of links: absolute and self-relative. An absolute link contains the absolute address of the entry to which it points. A self-relative link contains a displacement from the present queue entry. A queue is classified by the type of link it uses.

Because a queue contains redundant links, it is possible to create ill-formed queues. The VAX instructions produce UNPREDICTABLE results when used on ill-formed queues or on queues with overlapping entries.

3.9.1 Absolute Queues

Absolute queues use absolute addresses as links. Queue entries are linked by a pair of longwords.

The first (lowest addressed) longword is the forward link; it specifies the address of the succeeding queue entry. The second (highest addressed) longword is the backward link; it specifies the address of the preceding queue entry. A queue is specified by a queue header that is identical to a pair of queue linkage longwords. The forward link of the header is the address of the entry termed the head of the queue. The backward link of the header is the address of the entry termed the tail of the queue. The forward link of the tail points to the header.

Two general operations can be performed on queues: insertion of entries and removal of entries. Operations at the head or tail are always valid because the queue header is always present. Operations elsewhere in the queue depend on specific entries being present and may become invalid if another process is simultaneously performing operations on the queue. Therefore, if more than one process can perform operations on a queue simultaneously, insertions and removals should be done only at the head or tail of the queue. If only one process (or one process at a time) can perform operations on a queue, insertions and removals can be made at other than the head or tail of the queue.

Two instructions are provided for manipulating absolute queues: INSQUE and REMQUE. INSQUE inserts an entry specified by

an entry operand into the queue following the entry specified by the predecessor operand. REMQUE removes the entry specified by the entry operand. Queue entries can be on arbitrary byte boundaries. Both INSQUE and REMQUE are implemented as non-interruptible instructions.

3.9.2 *Self-Relative Queues*

Self-relative queues use displacements between queue entries as links. Queue entries are linked by a pair of longwords. The first longword (lowest addressed) is the forward link; it specifies the displacement of the succeeding queue entry from the present entry. The second longword (highest addressed) is the backward link; it specifies the displacement of the preceding queue entry from the present entry. A queue is specified by a queue header, which also consists of two longword links.

Four operations can be performed on self-relative queues: insert at head, insert at tail, remove from head, and remove from tail. Furthermore, these operations are interlocked to allow cooperating processes in a multiprocessor system to access a shared list without additional synchronization. Queue entries must be quadword aligned. A hardware-supported, interlocked memory access mechanism is used to read the queue header. Bit ⟨0⟩ of the queue header is used as a secondary interlock and is set when the queue is being accessed.

If an interlocked queue instruction encounters the secondary interlock set, then, in the presence of no exception conditions, it terminates after setting the condition codes to indicate failure to gain access to the queue. If the secondary interlock bit is not set, then the interlocked queue instruction sets it during its operation and clears it at instruction completion. This prevents other interlocked queue instructions from operating on the same queue.

If an interlocked queue instruction encounters both the secondary interlock set and an exception condition resulting from instruction execution, then it is UNPREDICTABLE whether the exception will occur or the instruction will terminate after setting the condition codes.

Chapter 5 discusses the results when multiple exceptions occur.

INSQHI — Insert Entry into Queue at Head, Interlocked

Format

```
opcode        entry.ab, header.aq
```

Operation

```
! Must have write access to header.
! Header must be quadword aligned.
! Header cannot be equal to entry.
tmp1 ← (header){interlocked}; ! Acquire hardware interlock.
                              ! tmp1<2:1> must be zero.
if tmp1<0> EQLU 1 then
   begin
   (header){interlocked} ← tmp1;
   ! Release hardware lock.
   {set condition codes and terminate instruction};
   end;

! Release hardware lock, and set secondary lock:
(header){interlocked} ← tmp1 OR 1;

If {all memory accesses can be completed} then
   ! Check if following addresses can be written
   ! without causing a memory management exception:
   !        entry
   !        header + tmp1
   ! Also, check for quadword alignment.
   begin
   tmp2 ← header - entry;
   ! Forward link of entry:
   (entry) ← tmp1 + tmp2;
   ! Backward link of entry:
   (entry + 4) ← tmp2;
   ! Backward link of successor:
   (header + tmp1 + 4) ← -tmp1 - tmp2;
   ! Get hardware interlock:
   {read (header){interlocked}};
   ! Forward link of header, release interlocks:
   (header){interlocked} ← -tmp2;
   end;
else
   begin
   ! Get hardware interlock:
   {read (header){interlocked}};
   ! Release all interlocks:
   (header){interlocked} ← tmp1;
   {backup instruction};
   {initiate fault};
   end;
```

Condition codes

```
if {secondary interlock was clear} then
   begin
   N ← 0;
   Z ← (entry) EQL (entry+4);        ! First entry in queue.
   V ← 0;
   C ← 0;
   end;
else
   begin
   N ← 0;
   Z ← 0;
   V ← 0;
   C ← 1;                    ! Secondary interlock failed.
   end;
```

Exceptions

reserved operand

Opcodes

5C INSQHI Insert Entry into Queue at Head, Interlocked

Description

The entry specified by the entry operand is inserted into the queue following the header. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise, the Z-bit is cleared. The insertion is a non-interruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, even in a multiprocessor environment. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory-management exception occurs, the queue is left in a consistent state (see Chapters 4 and 5). If the instruction fails to acquire the secondary interlock, then, in the presence of no exception conditions, the instruction sets condition codes and terminates.

Notes

1. Because the insertion is non-interruptible, processes running in kernel mode can share queues with interrupt service routines (see Chapters 4, 5, and 6).
2. The INSQHI, INSQTI, REMQHI, and REMQTI instructions are implemented such that cooperating software processes in a multiprocessor may access a shared list without additional synchronization.

3. To set a software interlock realized with a queue, the following code can be used:

```
INSERT: INSQHI  ...          ; Attempt to insert entry.
        BEQL    1$           ; If queue was empty, branch.
        BCS     INSERT       ; If queue was locked, retry.
        CALL    WAIT(...)    ; If entry was queued, wait.
1$:
```

4. During access validation, any access that cannot be completed results in a memory management exception even though the queue insertion is not started.
5. The following are reserved operands: entry or header, when either is an address that is not quadword aligned (if its address bits⟨2:0⟩ NEQU 0); (header), when (header)⟨2:1⟩ is not zero; header and entry, when they are equal. In the last case, the queue is not altered.
6. If the instruction encounters both the secondary interlock set and an exception condition resulting from execution, then it is UNPREDICTABLE whether the exception will occur or the instruction will terminate after setting the condition codes.

INSQTI — Insert Entry into Queue at Tail, Interlocked

Format

```
opcode        entry.ab, header.aq
```

Operation

```
!must have write access to header.
!header must be quadword aligned.
!header cannot be equal to entry.
tmp1 ← (header){interlocked};     !acquire hardware interlock
                                  !tmp1<2:1> must be zero.

if tmp1<0> EQLU 1 then
   begin
   (header){interlocked} ← tmp1; !release hardware interlock
   {set condition codes and terminate instruction};
   end;
else
   begin
   ! Set secondary interlock; release hardware interlock:
   (header){interlocked} ← tmp1 OR 1;
   If {all memory accesses can be completed} then
      !check if the following addresses can be written
      !without causing a memory management exception:
      !        entry
      !        header + (header + 4)
      !Also, check for quadword alignment
         begin
         tmp2 ← (header + 4);
         tmp3 ← header - entry;
         ! Forward link of entry:
         (entry) ← tmp3;
         ! Backward link of entry:
         (entry + 4) ← tmp2 + tmp3;
         if {tmp2 NEQU 0} then
            (header+tmp2) ← -tmp3 - tmp2
         else
            ! Forward link of predecessor:
            tmp1 ← -tmp3 - tmp2;
         ! Backward link of header:
         (header+4) ← -tmp3;
         ! Get hardware interlock:
         {read (header){interlocked}};
         ! Release interlocks:
         (header){interlocked} ← tmp1;
         end;
   else
      begin
      ! Get hardware interlocks:
      {read (header){interlocked}};
```

```
        ! Release interlocks:
        (header){interlocked} ← tmp1;
        {backup instruction};
        {initiate fault};
        end;
     end;
```

Condition codes

```
if {secondary interlock was clear} then
   begin
   N ← 0;
   Z ← (entry) EQL (entry+4);        !first entry in queue
   V ← 0;
   C ← 0;
   end;
else
   begin
   N ← 0;
   Z ← 0;
   V ← 0;
   C ← 1;                !secondary interlock failed
   end;
```

Exceptions

reserved operand

Opcodes

5D INSQTI Insert Entry into Queue at Tail, Interlocked

Description

The entry specified by the entry operand is inserted into the queue preceding the header. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise, the Z-bit is cleared. The insertion is a non-interruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, even in a multiprocessor environment. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 4 and 5). If the instruction fails to acquire the secondary interlock, then, in the presence of no exception conditions, the instruction sets condition codes and terminates.

Notes

1. Because the insertion is non-interruptible, processes running in kernel mode can share queues with interrupt service routines (see Chapters 4, 5, and 6).
2. The INSQHI, INSQTI, REMQHI, and REMQTI instructions are implemented such that cooperating software processes in

a multiprocessor may access a shared list without additional synchronization.

3. To set a software interlock realized with a queue, the following code can be used:

```
INSERT: INSQHI  ...          ; Attempt to insert entry.
        BEQL    1$           ; If queue was empty, branch.
        BCS     INSERT       ; If queue was locked, retry.
        CALL    WAIT(...)    ; If entry was queued, wait.
1$:
```

4. During access validation, any access that cannot be completed results in a memory management exception even though the queue insertion is not started.
5. The following are reserved operands: entry, header, or (header + 4), when any one is an address that is not quadword aligned (if its address bits⟨2:0⟩ NEQU 0); (header), when (header)⟨2:1⟩ is not zero; header and entry, when they are equal. In the last case, the queue is not altered.
6. If the instruction encounters both the secondary interlock set and an exception condition resulting from execution, then it is UNPREDICTABLE whether the exception will occur or the instruction will terminate after setting the condition codes.

INSQUE Insert Entry in Queue

Format

```
opcode        entry.ab, pred.ab
```

Operation

```
If {all memory accesses can be completed} then
   begin
   (entry) ← (pred);        !forward link of entry
   (entry + 4) ← pred;      !backward link of entry
   ((pred) + 4) ← entry;    !backward link of successor
   (pred) ← entry;          !forward link of predecessor
   end;
else
   begin
   {backup instruction};
   {initiate fault};
   end;
```

Condition codes

```
N ← (entry) LSS (entry+4);
Z ← (entry) EQL (entry+4);        !first entry in queue
V ← 0;
C ← (entry) LSSU (entry+4);
```

Exceptions

none

Opcodes

0E INSQUE Insert Entry in Queue

Description

The entry specified by the entry operand is inserted into the queue following the entry specified by the predecessor operand. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise, the Z-bit is cleared. The insertion is a non-interruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 4 and 5).

Notes

1. Three types of insertion can be performed by the appropriate choice of predecessor operand:
   - Insert at head

     ```
     INSQUE  entry,h      ;h is queue head
     ```

   - Insert at tail

     ```
     INSQUE  entry,@h+4   ;h is queue head
                          ;(Note "@" in this case only)
     ```

- Insert after arbitrary predecessor

```
INSQUE  entry,p          ;p is predecessor
```

2. Because the insertion is non-interruptible, processes running in kernel mode can share queues with interrupt service routines (see Chapters 4, 5, and 6).
3. The INSQUE and REMQUE instructions are implemented such that cooperating software processes in a single processor may access a shared list without additional synchronization if the insertions and removals are only at the head or tail of the queue.
4. To set a software interlock realized with a queue, the following code can be used:

```
   INSQUE  ...              ;was queue empty?
   BEQL    1$               ;yes
   CALL    WAIT(...)        ;no, wait
1$:
```

5. During access validation, any access that cannot be completed results in a memory management exception, even though the queue insertion is not started.

REMQHI Remove Entry from Queue at Head, Interlocked

Format

```
opcode        header.aq, addr.wl
```

Operation

```
!must have write access to header.
!header must be quadword aligned.
!header cannot equal address of addr.
tmp1 ← (header){interlocked}; !acquire hardware interlock.
                              !tmp1<2:1> must be zero.

if tmp1<0> EQLU 1 then
   begin
   ! Release hardware interlock:
   (header){interlocked} ← tmp1;
   {set condition codes and terminate instruction};
   end;
else
   begin
   ! Set secondary interlock and release hardware interlock:
   (header){interlocked} ← tmp1 OR 1;
   if {all memory accesses can be completed} then
      !check if the following can be done without
      !causing a memory management exception:
      !write addr operand
      !read contents of header + tmp1 {if tmp1 NEQU 0}
      !write into header + tmp1 + (header + tmp1)
      !                              {if tmp1 NEQU 0}
      !Also, check for quadword alignment
      begin
      tmp2 ← header + tmp1;
      addr ← tmp2;
      if {tmp1 EQL 0} then tmp3 ← header
         else tmp3 ← tmp2 + (tmp2);
      ! Backward link of successor:
      (tmp3 + 4) ← header - tmp3;
      ! Get hardware interlock:
      {read (header){interlocked}};
      ! Forward link of header and release all interlocks:
      (header){interlocked} ← tmp3 - header;
      end;
   else
      begin
      ! Get hardware interlock:
      {read (header){interlocked}};
      ! Release interlocks:
      (header){interlocked} ← tmp1;
      {backup instruction};
      {initiate fault};
```

```
      end;
   end;
```

Condition codes

```
if {secondary interlock was clear} then
   begin
   N ← 0;
   Z ← (header) EQL 0;    !queue empty after removal
   V ← tmp1 EQL 0;        !no entry to remove
   C ← 0;
   end;
else
   begin
   N ← 0;
   Z ← 0;
   V ← 1;                 !did not remove anything
   C ← 1;                 !secondary interlock failed
   end;
```

Exceptions

reserved operand

Opcodes

5E REMQHI Remove Entry from Queue at Head, Interlocked

Description

If the secondary interlock is clear, the queue entry following the header is removed from the queue and the address operand is replaced by the address of the entry removed. If the queue was empty prior to this instruction or if the secondary interlock failed, the condition code V-bit is set; otherwise, it is cleared.

If the interlock succeeded and the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise, the Z-bit is cleared. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, even in a multiprocessor environment. The removal is a non-interruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 4 and 5). If the instruction fails to acquire the secondary interlock, then, in the presence of no exception conditions, the instruction sets condition codes and terminates.

Notes

1. Because the removal is non-interruptible, processes running in kernel mode can share queues with interrupt service routines (see Chapters 4, 5, and 6).
2. The INSQHI, INSQTI, REMQHI, and REMQTI instructions are implemented such that cooperating software processes in

a multiprocessor may access a shared list without additional synchronization.

3. To release a software interlock realized with a queue, the following code can be used:

```
1$:     REMQHI  ...             ;removed last?
        BEQL    2$              ;yes
        BCS     1$              ;try removing again
        CALL    ACTIVATE(...)   ;activate other waiters
2$:
```

4. To remove entries until the queue is empty, the following code can be used:

```
1$:     REMQHI  ...             ;anything removed?
        BVS     2$              ;no
        .
        process removed entry
        .
        BR      1$
        .
        .
2$:     BCS     1$              ;try removing again
        queue empty
```

5. During access validation, any access that cannot be completed results in a memory management exception even though the queue removal is not started.
6. The following situations result in reserved operand exceptions: header or (header + (header)), when either is an address that is not quadword aligned (its address bits⟨2:0⟩ NEQU 0); (header), when (header)⟨2:1⟩ is not zero; the header address operand and the address of the addr operand, when they are equal. In the last case, the queue is not altered.
7. If the instruction encounters both the secondary interlock set and an exception condition resulting from execution, then it is UNPREDICTABLE whether the exception will occur or the instruction will terminate after setting the condition codes.

REMQTI Remove Entry from Queue at Tail, Interlocked

Format

```
opcode  header.aq, addr.wl
```

Operation

```
!must have write access to header.
!header must be quadword aligned.
!header cannot equal address of addr.
tmp1 ← (header){interlocked}; !acquire hardware interlock.
                              !tmp1<2:1> must be zero.

if tmp1<0> EQLU 1 then
   begin
   ! Release hardware interlock:
   (header){interlocked} ← tmp1;
   {set condition codes and terminate instruction};
   end;
else
   begin
   ! Set secondary interlock and release hardware interlock:
   (header){interlocked} ← tmp1 OR 1;

   If {all memory accesses can be completed} then
      !check if the following can be done without
      !causing a memory management exception:
      !write addr operand
      !read contents of header + (header + 4)
      !                       {if tmp1 NEQU 0}
      !write into header + (header + 4)
      !  + (header + 4 + (header + 4)) {if tmp1 NEQU 0}
      !Also, check for quadword alignment
      begin
      addr ← header + (header + 4);
      tmp2 ← addr + (addr + 4);
      ! Backward link of header:
      (header + 4) ← tmp2 - header;
      ! Save tmp1 to set Z correctly:
      tmp3 ← tmp1;
      if tmp2 EQL header then tmp1 ← 0
      else
         ! Forward link of predecessor:
         (tmp2) ← header - tmp2;
      {read (header){interlocked}};
      ! Release interlocks:
      (header){interlocked} ← tmp1;
      end;
   else
      begin
      ! Get hardware interlock:
```

```
        {read (header){interlocked}};
        ! Release interlocks:
        (header){interlocked} ← tmp1;
        {backup instruction};
        {initiate fault};
        end;
     end;
```

Condition codes

```
     if {secondary interlock was clear} then
        begin
        N ← 0;
        Z ← (header + 4) EQL 0;  !queue empty after removal
        V ← tmp3 EQL 0;          !no entry to remove
        C ← 0;
        end;
     else
        begin
        N ← 0;
        Z ← 0;
        V ← 1;              !did not remove anything
        C ← 1;              !secondary interlock failed
        end;
```

Exceptions reserved operand

Opcodes 5F REMQTI Remove Entry from Queue at Tail, Interlocked

Description If the secondary interlock is clear, the queue entry preceding the header is removed from the queue and the address operand is replaced by the address of the entry removed. If the queue was empty prior to this instruction or if the secondary interlock failed, the condition code V-bit is set; otherwise, it is cleared.

If the interlock succeeded and the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise, the Z-bit is cleared. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, even in a multiprocessor environment. The removal is a non-interruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 4 and 5). If the instruction fails to acquire the secondary interlock, then, in the presence of no exception conditions, the instruction sets condition codes and terminates.

1. Because the removal is non-interruptible, processes running in kernel mode can share queues with interrupt service routines (see Chapters 4, 5, and 6).
2. The INSQHI, INSQTI, REMQHI, and REMQTI instructions are implemented such that cooperating software processes in a multiprocessor may access a shared list without additional synchronization.
3. To release a software interlock realized with a queue, the following code can be used:

```
1$:     REMQTI  ...             ;removed last?
        BEQL    2$              ;yes
        BCS     1$              ;try removing again
        CALL    ACTIVATE(...)   ;Activate other waiters
2$:
```

4. To remove entries until the queue is empty, the following code can be used:

```
1$:     REMQTI  ...             ;anything removed?
        BVS     2$              ;no
        .
        process removed entry
        .
        BR      1$
        .
        .
2$:     BCS     1$              ;try removing again
        queue empty
```

5. During access validation, any access that cannot be completed results in a memory management exception even though the queue removal is not started.
6. The following situations result in reserved operands exceptions: header, (header + 4), or (header + (header + 4) + 4), when any one is an address that is not quadword aligned (if its address bits⟨2:0⟩ NEQU 0); (header), when (header)⟨2:1⟩ is not zero; the header address operand and the address of the addr operand, when they are equal. In the last case, the queue is not altered.
7. If the instruction encounters both the secondary interlock set and an exception condition resulting from execution, then it is UNPREDICTABLE whether the exception will occur or the instruction will terminate after setting the condition codes.

REMQUE Remove Entry from Queue

Format

```
opcode        entry.ab,addr.wl
```

Operation

```
if {all memory accesses can be completed} then
   begin
   ((entry+4)) ← (entry);      !forward link of predecessor
   ((entry)+4) ← (entry +4);   !backward link of successor
   addr ← entry;
   end;
else
   begin
   {backup instruction};
   {initiate fault};
   end;
```

Condition codes

```
N ← (entry) LSS (entry+4);
Z ← (entry) EQL (entry+4);        !queue empty
V ← entry EQL (entry+4);          !no entry to remove
C ← (entry) LSSU (entry+4);
```

Exceptions

none

Opcodes

0F REMQUE Remove Entry from Queue

Description

The queue entry specified by the entry operand is removed from the queue. The address operand is replaced by the address of the entry removed. If there was no entry in the queue to be removed, the condition code V-bit is set; otherwise, it is cleared. If the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise, the Z-bit is cleared. The removal is a non-interruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state (see Chapters 4 and 5).

Notes

1. Three types of removal can be performed by suitable choice of entry operand:

   - Remove at head

     ```
     REMQUE  @h,addr          ;h is queue header
     ```

   - Remove at tail

     ```
     REMQUE  @h+4,addr        ;h is queue header
     ```

- Remove arbitrary entry

```
REMQUE  entry,addr
```

2. Because the removal is non-interruptible, processes running in kernel mode can share queues with interrupt service routines (see Chapters 4, 5, and 6).
3. The INSQUE and REMQUE instructions are implemented such that cooperating software processes in a single processor may access a shared list without additional synchronization if the insertions and removals are only at the head or tail of the queue.
4. To release a software interlock realized with a queue, the following code can be used:

```
        REMQUE  ...             ;queue empty?
        BEQL    1$              ;yes
        CALL    ACTIVATE(...)   ;Activate other waiters
1$:
```

5. To remove entries until the queue is empty, the following code can be used:

```
1$:     REMQUE  ...             ;anything removed?
        BVS     EMPTY           ;no
        .
        .
        .
        BR      1$
```

6. During access validation, any access that cannot be completed results in a memory management exception even though the queue removal is not started.

3.10 Floating-Point Instructions

The floating-point instructions operate on four data types, termed F_floating (32 bits of precision), D_floating (64 bits of precision), G_floating (64 bits of precision), and H_floating (128 bits of precision). In most cases, VAX processors are designed to optimize the performance of these instructions. However, H_floating instructions and the EMOD and POLY instructions for all data types are optional to implement and so can be omitted.

Execution of an omitted instruction results in a reserved instruction fault. Omitted instructions are emulated by Digital-supported operating systems. Emulation software may allocate and use current-mode stack space when executed. For more detail, refer to Chapter 11.

3.10.1 Representation

Mathematically, a floating-point number may be defined as having the form

```
(+ or -) (2**K)*f
```

where K is an integer and f is a non-negative fraction. For a non-vanishing number, K and f are uniquely determined by imposing the condition

```
1/2 LEQ f LSS 1
```

The fractional factor, f, of the number is then said to be binary normalized. For the number zero, f must be assigned the value 0, and the value of K is indeterminate.

The VAX floating-point data formats are derived from this mathematical representation for floating-point numbers. Four types of floating-point data are provided: the two standard PDP-11 formats (F_floating and D_floating) and two extended range formats (G_floating and H_floating). Single-precision, or floating, data is 32 bits long. Double-precision, or D_floating, data is 64 bits long. Extended range double-precision, or G_floating, data is 64 bits long. Extended range quadruple-precision, or H_floating, data is 128 bits long. Sign magnitude notation is used.

The most significant bit of the floating-point data is the sign bit: 0 for positive and 1 for negative.

The fractional factor f is assumed normalized, so that its most significant bit must be 1. This 1 is the "hidden" bit: it is not stored in the data word, but of course the hardware restores it before carrying out arithmetic operations. The F_floating and D_floating data types use 23 and 55 bits, respectively, for f, which with the hidden bit imply effective significance of 24 bits and 56 bits for arithmetic operations. The extended range data types, G_floating and H_floating, use 52 and 112 bits, respectively, for f, which with the hidden bit imply effective significance of 53 and 113 bits for arithmetic operations.

In the F_floating and D_floating data types, 8 bits are reserved for the storage of the exponent K in excess 128 notation. Thus, exponents from −128 to +127 could be represented, in biased form, by 0 to 255. For reasons given below, a biased EXP of 0 (true exponent of −128) is reserved for floating-point zero. Thus, for the F_floating and D_floating data types, exponents are restricted to the range −127 to +127, inclusive, or in excess 128 notation, 1 to 255.

In the G_floating data type, 11 bits are reserved for the storage of the exponent in excess 1024 notation. Thus, exponents are restricted to −1023 to +1023, inclusive (in excess notation, 1 to 2047). In the H_floating data type, 15 bits are reserved for the storage of the exponent in excess 16384 notation. Thus, exponents are restricted to −16383 to +16383, inclusive (in excess notation, 1 to 32767). A biased exponent of 0 is reserved for floating-point zero.

Because of the hidden bit, the fractional factor is not available to distinguish between zero and non-zero numbers whose fractional factor is exactly 1/2. Therefore, VAX architecture reserves a sign-exponent field of 0 for this purpose. Any positive, floating-point number with biased exponent of 0 is treated as if it were an exact 0 by the floating-point instruction set. In particular, a floating-point operand, whose bits are all zeros, is treated as zero; this is the format generated by all floating-point instructions for which the result is zero.

A reserved operand is defined to be any bit pattern with a sign bit of 1 and a biased exponent of 0. In VAX architecture, all floating-

point instructions generate a fault if a reserved operand is encountered. Scalar floating-point instructions (which are described in the following sections) never generate a reserved operand. However, vector floating-point instructions may generate reserved operands (see section 12.8).

3.10.2 *Overview of the Instruction Set*

The VAX architecture has the standard arithmetic operations ADD, SUB, MUL, and DIV implemented for all four floating-point data types. The results of these operations are always rounded, as described in the following section on accuracy. The architecture has, in addition, two composite operations, EMOD and POLY, also implemented for all four floating-point data types. EMOD generates a product of two operands and then separates the product into its integer and fractional terms. POLY evaluates a polynomial, given the degree, the argument, and pointer to a table of coefficients. Details on the operation of EMOD and POLY are given in their respective descriptions. All of these instructions are subject to the rounding errors associated with floating-point operations as well as to exponent overflow and underflow. Accuracy is discussed in the next section, and exceptions are discussed in Chapter 5.

The VAX architecture also has a complete set of instructions for conversion from integer arithmetic types (byte, word, longword) to all floating types (F_floating, D_floating, G_floating, H_floating) and also for floating types to integer arithmetic types. VAX also has a set of instructions for conversion between all the floating types except between D_floating and G_floating. Many of these instructions are exact, in the sense defined in the section on accuracy to follow. A few instructions, however, may generate rounding error, floating overflow, or floating underflow, or they may induce integer overflow. Details are given in the description of the CVT instructions.

There is a class of move-type instructions that is always exact: MOV, NEG, CLR, CMP, and TST. And, finally, there is the ACB (add, compare, and branch) instruction, which is subject to rounding errors, overflow, and underflow.

All VAX floating-point instructions fault if a reserved operand is encountered. Floating-point instructions also fault on the occur-

rence of floating overflow or divide by zero. PSL⟨FU⟩ is available to enable or disable an exception on underflow. If PSL⟨FU⟩ is clear, no exception occurs on underflow, and zero is returned as the result. If PSL⟨FU⟩ is set, a fault occurs on underflow. Further details on the actions taken if any of these exceptions occurs are included in the descriptions of the instructions and are completely discussed in Chapter 5.

3.10.3 *Accuracy*

General comments on the accuracy of the VAX floating-point instruction set are presented here. The descriptions of the individual instructions may include additional details on the accuracy at which they operate.

An instruction is defined to be exact if its result, extended on the right by an infinite sequence of zeros, is identical to that of an infinite precision calculation involving the same operands. The a priori accuracy of the operands is thus ignored. For all arithmetic operations except DIV, a zero operand implies that the instruction is exact. The same statement holds for DIV if the zero operand is the dividend. But if the zero operand is the divisor, division is undefined and the instruction faults.

For non-zero, floating-point operands, the fractional factor is binary normalized with 24 bits for single precision (F_floating) and 56 bits for double precision (D_floating); the fractional factor is binary normalized with 53 bits for extended range double precision (G_floating) and 113 bits for extended range quadruple precision (H_floating).

Note that an arithmetic result is exact if no non-zero bits are lost in chopping the infinite precision result to the data length to be stored. Chopping means that the 24 (F_floating), 56 (D_floating), 53 (G_floating), or 113 (H_floating) high-order bits of the normalized fractional factor of a result are stored; the rest of the bits are discarded. The first bit lost in chopping is referred to as the "rounding" bit. The value of a rounded result is related to the chopped result as follows:

1. If the rounding bit is 1, the rounded result is the chopped result incremented by an LSB (least significant bit).

2. If the rounding bit is 0, the rounded and chopped results are identical.

All VAX processors implement rounding to produce results identical to the results produced by the following algorithm. Add a 1 to the rounding bit and propagate the carry if it occurs. Note that a renormalization may be required after rounding takes place. If this happens, the new rounding bit will be 0. Therefore, renormalization can happen only once. The following statements summarize the relations among chopped, rounded, and true (infinite precision) results:

1. If a stored result is exact

```
rounded value = chopped value = true value
```

2. If a stored result is not exact, its magnitude
   - Is always less than that of the true result for chopping
   - Is always less than that of the true result for rounding if the rounding bit is zero
   - Is greater than that of the true result for rounding if the rounding bit is one

3.10.4 *Programming Considerations*

To be consistent with the floating-point instruction set that faults on reserved operands, software-implemented floating-point functions (the absolute function, for example) should verify that the input operand(s) is (are) not reserved. An easy way to do this is a floating move or test of the input operand(s).

To facilitate high-speed implementations of the floating-point instruction set, certain restrictions are placed on the addressing mode combinations usable within a single floating-point instruction. These combinations involve the logically inconsistent simultaneous use of a value as both a floating-point operand and an address. Specifically, if within the same instruction the content of register Rn is used as both a part of a floating-point input operand (operand type .rf, .rd, .rg, .rh, .mf, .md, .mg, or .mh) and as an address in an addressing mode that modifies Rn (autoincrement, autodecrement, or autoincrement deferred), the value of the floating-point operand is UNPREDICTABLE.

ADD Add

Format

```
opcode add.rx, sum.mx                2 operand
opcode add1.rx, add2.rx, sum.wx      3 operand
```

Operation

```
sum ← sum + add;                     !2 operand
sum ← add1 + add2;                   !3 operand
```

Condition codes

```
N ← sum LSS 0;
Z ← sum EQL 0;
V ← 0;
C ← 0;
```

Exceptions

```
floating overflow
floating underflow
reserved operand
```

Opcodes

40	ADDF2	Add F_floating 2 Operand
41	ADDF3	Add F_floating 3 Operand
60	ADDD2	Add D_floating 2 Operand
61	ADDD3	Add D_floating 3 Operand
40FD	ADDG2	ADD G_floating 2 Operand
41FD	ADDG3	ADD G_floating 3 Operand
60FD	ADDH2	ADD H_floating 2 Operand
61FD	ADDH3	ADD H_floating 3 Operand

Description

In 2 operand format, the addend operand is added to the sum operand and the sum operand is replaced by the rounded result. In 3 operand format, the addend 1 operand is added to the addend 2 operand and the sum operand is replaced by the rounded result.

Notes

1. On a reserved operand fault, the sum operand is unaffected and the condition codes are UNPREDICTABLE.
2. On floating underflow, a fault occurs if PSL⟨FU⟩ is set. Zero is stored as the result of floating underflow only if PSL⟨FU⟩ is clear. On a floating underflow fault, the sum operand is unaffected. If PSL⟨FU⟩ is clear, the sum operand is replaced by 0 and no exception occurs.
3. On floating overflow, the instruction faults. The sum operand is unaffected, and the condition codes are UNPREDICTABLE.
4. The ADDHx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

CLR Clear

Format

```
opcode dst.wx
```

Operation

```
dst ← 0;
```

Condition codes

```
N ← 0;
Z ← 1;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

D4	CLRF	Clear F_floating
7C	CLRG	Clear G_floating
	CLRD	Clear D_floating
7CFD	CLRH	Clear H_floating

Description

The destination operand is replaced by 0.

Notes

1. *CLRx dst* is equivalent to *MOVx #0, dst*, but is 5 (F_floating), 9 (D_floating or G_floating), or 17 (H_floating) bytes shorter.
2. The CLRH instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

CMP — Compare

Format

```
opcode src1.rx, src2.rx
```

Operation

```
src1 - src2;
```

Condition codes

```
N ← src1 LSS src2;
Z ← src1 EQL src2;
V ← 0;
C ← 0;
```

Exceptions

```
reserved operand
```

Opcodes

51	CMPF	Compare F_floating
71	CMPD	Compare D_floating
51FD	CMPG	Compare G_floating
71FD	CMPH	Compare H_floating

Description

The source 1 operand is compared with the source 2 operand. The only action is to affect the condition codes.

Notes

1. On a reserved operand fault, the condition codes are UNPREDICTABLE.
2. The CMPH instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

CVT Convert

Format

```
opcode src.rx, dst.wy
```

Operation

```
dst ← conversion of src;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {integer overflow};
C ← 0;
```

Exceptions

```
integer overflow
floating overflow
floating underflow
reserved operand
```

Opcodes

4C	CVTBF	Convert Byte to F_floating
4D	CVTWF	Convert Word to F_floating
4E	CVTLF	Convert Long to F_floating
6C	CVTBD	Convert Byte to D_floating
6D	CVTWD	Convert Word to D_floating
6E	CVTLD	Convert Long to D_floating
4CFD	CVTBG	Convert Byte to G_floating
4DFD	CVTWG	Convert Word to G_floating
4EFD	CVTLG	Convert Long to G_floating
6CFD	CVTBH	Convert Byte to H_floating
6DFD	CVTWH	Convert Word to H_floating
6EFD	CVTLH	Convert Long to H_floating
48	CVTFB	Convert F_floating to Byte
49	CVTFW	Convert F_floating to Word
4A	CVTFL	Convert F_floating to Long
4B	CVTRFL	Convert Rounded F_floating to Long
68	CVTDB	Convert D_floating to Byte
69	CVTDW	Convert D_floating to Word
6A	CVTDL	Convert D_floating to Long
6B	CVTRDL	Convert Rounded D_floating to Long
48FD	CVTGB	Convert G_floating to Byte
49FD	CVTGW	Convert G_floating to Word
4AFD	CVTGL	Convert G_floating to Long
4BFD	CVTRGL	Convert Rounded G_floating to Long
68FD	CVTHB	Convert H_floating to Byte
69FD	CVTHW	Convert H_floating to Word
6AFD	CVTHL	Convert H_floating to Long
6BFD	CVTRHL	Convert Rounded H_floating to Long
56	CVTFD	Convert F_floating to D_floating
99FD	CVTFG	Convert F_floating to G_floating
98FD	CVTFH	Convert F_floating to H_floating
76	CVTDF	Convert D_floating to F_floating
32FD	CVTDH	Convert D_floating to H_floating

33FD	CVTGF	Convert G_floating to F_floating
56FD	CVTGH	Convert G_floating to H_floating
F6FD	CVTHF	Convert H_floating to F_floating
F7FD	CVTHD	Convert H_floating to D_floating
76FD	CVTHG	Convert H_floating to G_floating

Description

The source operand is converted to the data type of the destination operand, and the destination operand is replaced by the result. The form of the conversion is as follows:

CVTBF	exact	CVTHW	truncated
CVTBD	exact	CVTFL	truncated
CVTBG	exact	CVTRFL	rounded
CVTBH	exact	CVTDL	truncated
CVTWF	exact	CVTRDL	rounded
CVTWD	exact	CVTGL	truncated
CVTWG	exact	CVTRGL	rounded
CVTWH	exact	CVTHL	truncated
CVTLF	rounded	CVTRHL	rounded
CVTLD	exact	CVTFD	exact
CVTLG	exact	CVTFG	exact
CVTLH	exact	CVTFH	exact
CVTFB	truncated	CVTDF	rounded
CVTDB	truncated	CVTDH	exact
CVTGB	truncated	CVTGF	rounded
CVTHB	truncated	CVTGH	exact
CVTFW	truncated	CVTHF	rounded
CVTDW	truncated	CVTHD	rounded
CVTGW	truncated	CVTHG	rounded

Notes

1. Only CVTDF, CVTGF, CVTHF, CVTHD, and CVTHG can result in floating overflow fault. The destination operand is unaffected, and the condition codes are UNPREDICTABLE.
2. Only conversions with a floating-point source operand can result in a reserved operand fault. On a reserved operand fault, the destination operand is unaffected and the condition codes are UNPREDICTABLE.
3. Only conversions with an integer destination operand can result in integer overflow. On integer overflow, the destination operand is replaced by the low-order bits of the true result.
4. Only CVTGF, CVTHF, CVTHD, and CVTHG can result in floating underflow. If PSL⟨FU⟩ is set, a fault occurs. Zero is stored as the result of floating underflow only if PSL⟨FU⟩ is clear. On a floating underflow fault, the destination operand is unaffected. If PSL⟨FU⟩ is clear, the destination operand is replaced by 0 and no exception occurs.

5. When CVTRFL, CVTRDL, CVTRGL, and CVTRHL round, the rounding is done in sign magnitude, before conversion to two's complement.
6. The CVTxH, CVTHx, and CVTRHL instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

DIV — Divide

Format

```
opcode divr.rx, quo.mx                2 operand
opcode divr.rx, divd.rx, quo.wx       3 operand
```

Operation

```
quo ← quo / divr;                     !2 operand
quo ← divd / divr;                    !3 operand
```

Condition codes

```
N ← quo LSS 0;
Z ← quo EQL 0;
V ← 0;
C ← 0;
```

Exceptions

```
floating overflow
floating underflow
divide by zero
reserved operand
```

Opcodes

46	DIVF2	Divide F_floating 2 Operand
47	DIVF3	Divide F_floating 3 Operand
66	DIVD2	Divide D_floating 2 Operand
67	DIVD3	Divide D_floating 3 Operand
46FD	DIVG2	Divide G_floating 2 Operand
47FD	DIVG3	Divide G_floating 3 Operand
66FD	DIVH2	Divide H_floating 2 Operand
67FD	DIVH3	Divide H_floating 3 Operand

Description

In 2 operand format, the quotient operand is divided by the divisor operand and the quotient operand is replaced by the rounded result. In 3 operand format, the dividend operand is divided by the divisor operand and the quotient operand is replaced by the rounded result.

Notes

1. On a reserved operand fault, the quotient operand is unaffected and the condition codes are UNPREDICTABLE.
2. On floating underflow, a fault occurs if PSL⟨FU⟩ is set. Zero is stored as the result of floating underflow only if PSL⟨FU⟩ is clear. On a floating underflow fault, the quotient operand is unaffected. If PSL⟨FU⟩ is clear, the quotient operand is replaced by 0 and no exception occurs.
3. On floating overflow, the instruction faults. The quotient operand is unaffected, and the condition codes are UNPREDICTABLE.
4. On divide by zero, the quotient operand and condition codes are affected as in note 3 above.
5. The DIVHx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

EMOD Extended Multiply and Integerize

Format

```
EMODF and EMODD:
opcode mulr.rx, mulrx.rb, muld.rx, int.wl, fract.wx

EMODG and EMODH:
opcode mulr.rx, mulrx.rw, muld.rx, int.wl, fract.wx
```

Operation

```
int ← integer part of muld * {mulr'mulrx};
fract ← fractional part of muld * {mulr'mulrx};
```

Condition codes

```
N ← fract LSS 0;
Z ← fract EQL 0;
V ← {integer overflow};
C ← 0;
```

Exceptions

```
integer overflow
floating underflow
reserved operand
```

Opcodes

54	EMODF	Extended Multiply and Integerize F_floating
74	EMODD	Extended Multiply and Integerize D_floating
54FD	EMODG	Extended Multiply and Integerize G_floating
74FD	EMODH	Extended Multiply and Integerize H_floating

Description

The multiplier extension operand is concatenated with the multiplier operand to gain 8 (EMODD and EMODF), 11 (EMODG), or 15 (EMODH) additional low-order fraction bits. The low-order 5 or 1 bits of the 16-bit multiplier extension operand are ignored by the EMODG and EMODH instructions, respectively. The multiplicand operand is multiplied by the extended multiplier operand. The multiplication is such that the result is equivalent to the exact product truncated (before normalization) to a fraction field of 32 bits in F_floating, 64 bits in D_floating and G_floating, and 128 in H_floating. Regarding the result as the sum of an integer and fraction of the same sign, the integer operand is replaced by the integer part of the result; the fraction operand is replaced by the rounded fractional part of the result.

Notes

1. On a reserved operand fault, the integer operand and the fraction operand are unaffected. The condition codes are UNPREDICTABLE.
2. On floating underflow, a fault occurs if PSL⟨FU⟩ is set. The integer and fraction parts are replaced by 0 on the occurrence of floating underflow only if PSL⟨FU⟩ is clear. On a floating underflow fault, the integer and fraction parts are unaffected.

If PSL⟨FU⟩ is clear, the integer and fraction parts are replaced by 0 and no exception occurs.

3. On integer overflow, the integer operand is replaced by the low-order bits of the true result.
4. Floating overflow is indicated by integer overflow. Integer overflow is possible, however, in the absence of floating overflow.
5. The signs of the integer and the fraction are the same unless integer overflow results or the fraction is zero.
6. The integer and fraction are separated before the integer is converted to two's complement and before the fraction is rounded. Because rounding occurs after separation, the value of the fraction operand may be 1.
7. The EMODx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11.
8. The EMODx instructions are in the emulated-only group and are unlikely to be implemented in future VAX processors. Software developers are advised to avoid using these instructions.

MNEG Move Negated

Format

```
opcode src.rx, dst.wx
```

Operation

```
dst ← -src;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← 0;
C ← 0;
```

Exceptions

```
reserved operand
```

Opcodes

52	MNEGF	Move Negated F_floating
72	MNEGD	Move Negated D_floating
52FD	MNEGG	Move Negated G_floating
72FD	MNEGH	Move Negated H_floating

Description

The destination operand is replaced by the negative of the source operand.

Notes

1. On a reserved operand fault, the destination operand is unaffected and the condition codes are UNPREDICTABLE.
2. The MNEGH instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

MOV — Move

Format

```
opcode src.rx, dst.wx
```

Operation

```
dst ← src;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← 0;
C ← C;
```

Exceptions

```
reserved operand
```

Opcodes

50	MOVF	Move F_floating
70	MOVD	Move D_floating
50FD	MOVG	Move G_floating
70FD	MOVH	Move H_floating

Description

The destination operand is replaced by the source operand.

Notes

1. On a reserved operand fault, the destination operand is unaffected and the condition codes are UNPREDICTABLE.
2. The MOVH instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

MUL Multiply

Format

```
opcode mulr.rx, prod.mx                2 operand
opcode mulr.rx, muld.rx, prod.wx       3 operand
```

Operation

```
prod ← prod * mulr;                    !2 operand
prod ← muld * mulr;                    !3 operand
```

Condition codes

```
N ← prod LSS 0;
Z ← prod EQL 0;
V ← 0;
C ← 0;
```

Exceptions

```
floating overflow
floating underflow
reserved operand
```

Opcodes

44	MULF2	Multiply F_floating 2 Operand
45	MULF3	Multiply F_floating 3 Operand
64	MULD2	Multiply D_floating 2 Operand
65	MULD3	Multiply D_floating 3 Operand
44FD	MULG2	Multiply G_floating 2 Operand
45FD	MULG3	Multiply G_floating 3 Operand
64FD	MULH2	Multiply H_floating 2 Operand
65FD	MULH3	Multiply H_floating 3 Operand

Description

In 2 operand format, the product operand is multiplied by the multiplier operand and the product operand is replaced by the rounded result. In 3 operand format, the multiplicand operand is multiplied by the multiplier operand and the product operand is replaced by the rounded result.

Notes

1. On a reserved operand fault, the product operand is unaffected and the condition codes are UNPREDICTABLE.
2. On floating underflow, a fault occurs if PSL⟨FU⟩ is set. Zero is stored as the result of floating underflow only if PSL⟨FU⟩ is clear. On a floating underflow fault, the product operand is unaffected. If PSL⟨FU⟩ is clear, the product operand is replaced by 0 and no exception occurs.
3. On floating overflow, the instruction faults. The product operand is unaffected, and the condition codes are UNPREDICTABLE.
4. The MULHx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

POLY Polynomial Evaluation

Format

```
opcode arg.rf, degree.rw, tbladdr.ab, {R0-3.wl}  !POLYF
opcode arg.rd, degree.rw, tbladdr.ab, {R0-5.wl}  !POLYD
opcode arg.rg, degree.rw, tbladdr.ab, {R0-5.wl}  !POLYG
opcode arg.rh, degree.rw, tbladdr.ab,
       {R0-5.wl,-16(SP):-1(SP).wb}               !POLYH
```

Operation

```
tmp1 ← degree;
if tmp1 GTRU 31 then {initiate reserved operand fault};
tmp2 ← tbladdr;
tmp3 ← {(tmp2)+};     !tmp3 accumulates the partial result
                      !tmp3 is of type x
if POLYH then -(SP) ← arg;
while tmp1 GTRU 0 do
begin
   !Begin computation loop
   !tmp4 accumulates new partial result.
   !tmp3 has old partial result.
tmp4 ← {arg * tmp3};

   ! Perform multiply, and retain either 31 or 32 (POLYF),
   ! either 63 or 64 (for POLYD, POLYG), and either 127
   ! or 128 (for POLYH) most significant bits of the
   ! fraction by truncating the unnormalized product
   ! toward zero.  (The most significant bits in the
   ! product fraction will be 01 (binary) if the product
   ! fraction is LSS 1/2 and GEQ 1/4, and the most
   ! significant bit will be 1 otherwise.)  Use the
   ! result in the following add operation.

tmp4 ← tmp4 + (tmp2);
   ! Align fractions, perform add, and retain either  the
   ! 31 or 32 (for POLYF), 63 or 64 (for POLYD or POLYG),
   ! and 127 or 128 (for POLYH) most significant bits of
   ! the fraction by truncating the unnormalized result
   ! toward zero.  Normalize, and round to type x.  Check
   ! for overflow and underflow only after the combined
   ! multiply, add, normalize, round sequence.

if OVERFLOW then FLOATING OVERFLOW FAULT
if UNDERFLOW then
   begin
   if PSL<FU> EQL 1 then FLOATING UNDERFLOW FAULT;
   tmp4 ← 0;         !force result to 0;
   end;
tmp1 ← tmp1 - 1;
tmp2 ← tmp2 + {size of data type};
```

```
tmp3 ← tmp4;          !update partial result in tmp3
end;
if POLYF then
   begin
   R0 ← tmp3;
   R1 ← 0;
   R2 ← 0;
   R3 ← tmp2;
   end;
if POLYD or POLYG then
   begin
   R1'R0 ← tmp3;
   R2 ← 0;
   R3 ← tmp2;
   R4 ← 0;
   R5 ← 0;
   end;
if POLYH then
   begin
   SP ← SP + 16;
   R3'R2'R1'R0 ← tmp3;
   R4 ← 0;
   R5 ← tmp2;
   end;
```

Condition codes

```
N ← R0 LSS 0;
Z ← R0 EQL 0;
V ← 0;
C ← 0;
```

Exceptions

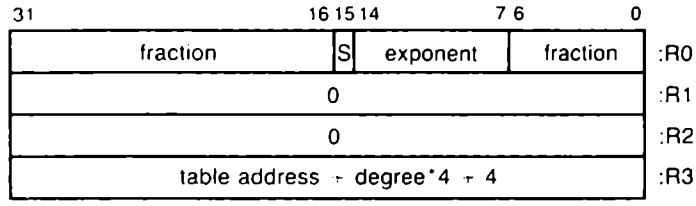
```
floating overflow
floating underflow
reserved operand
```

Opcodes

55	POLYF	Polynomial Evaluation F_floating
75	POLYD	Polynomial Evaluation D_floating
55FD	POLYG	Polynomial Evaluation G_floating
75FD	POLYH	Polynomial Evaluation H_floating

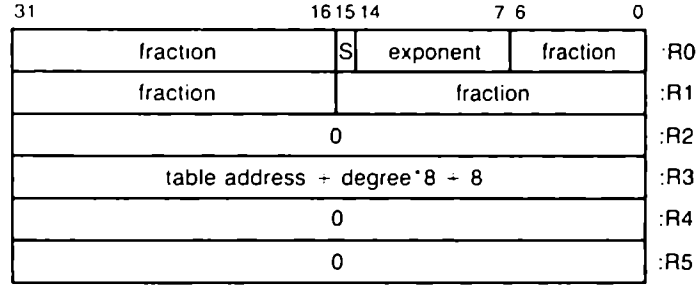
Description

The table address operand points to a table of polynomial coefficients. The coefficient of the highest order term of the polynomial is pointed to by the table address operand. The table is specified with lower order coefficients stored at increasing addresses. The data type of the coefficients is the same as the data type of the argument operand. The evaluation is carried out by Horner's method, and the contents of R0 (R1'R0 for POLYD and POLYG, R3'R2'R1'R0 for POLYH) are replaced by the result. The result computed is:



POLYF

Figure 3.3
Result Registers After POLYF



POLYD

Figure 3.4
Result Registers After POLYD

```
result = C[0]*x**0 + x*(C[1] + x*(C[2] + ... x*C[d]))
```

where x is the argument and d is the degree. The unsigned-word degree operand specifies the highest numbered coefficient to participate in the evaluation. POLYH requires four longwords on the stack to store arg in case the instruction is interrupted.

Example

To compute P(x) = C0 + C1*x + C2*x**2, where C0 = 1.0, C1 = .5, and C2 = .25:

```
         POLYF   X,#2,PTABLE
         .
         .
         .
PTABLE:  .FLOAT  0.25     ;C2
         .FLOAT  0.5      ;C1
         .FLOAT  1.0      ;C0
```

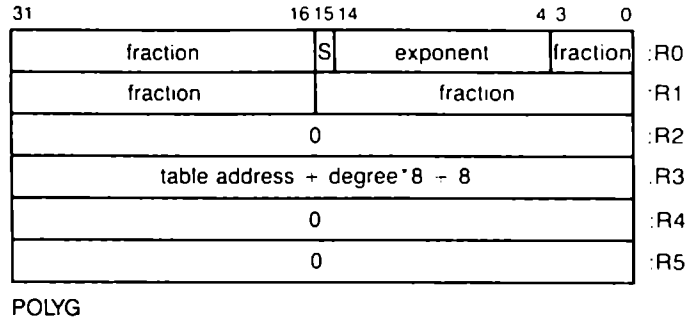


Figure 3.5
Result Registers After POLYG

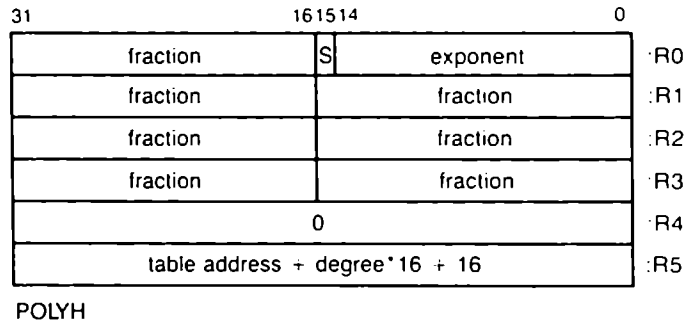


Figure 3.6
Result Registers After POLYH

Notes

1. After execution, the registers are as shown in Figures 3.3 through 3.6.
2. On a floating fault:
   - If PSL⟨FPD⟩ = 0, the instruction faults and all relevant side effects are restored to their original state.
   - If PSL⟨FPD⟩ = 1, the instruction is suspended and state is saved in the general registers as follows:

```
POLYF
R0 = tmp3          ! Partial result after iteration prior
                   !  to the one causing the overflow or
                   !  underflow.
R1 = arg
R2<7:0> = tmp1     ! Number of iterations remaining.
R2<31:8> = impl    ! Implementation-dependent value.
R3 = tmp2          ! Address of the table entry causing
                   !  the exception.
```

```
POLYD and POLYG
R1'R0 = tmp3     ! Partial result after iteration prior
                 !  to the one causing the overflow or
                 !  underflow.
R2<7:0> = tmp1   ! Number of iterations remaining.
R2<31:8> = impl  ! Implementation-dependent value.
R3 = tmp2        ! Address of the table entry causing
                 !  the exception.
R5'R4 = arg

POLYH
R3'R2'R1'R0 = tmp3  ! Partial result after iteration
                    !  prior to the one causing the
                    !  overflow or underflow.
R4<7:0> = tmp1   ! Number of iterations remaining.
R4<31:8> = impl  ! Implementation-dependent value.
R5 = tmp2        ! Address of the table entry causing
                 !  the exception.
```

Arg is saved on the stack in use during the faulting instruction. Implementation-dependent information is saved to allow the instruction to continue after possible scaling of the coefficients and partial result by a fault handler.

3. If the unsigned-word degree operand is 0 and the argument is not a reserved operand, the result is C[0]. If the degree is 0 and either the argument or C[0] is a reserved operand, a reserved operand fault occurs.
4. If the unsigned-word degree operand is greater than 31, a reserved operand fault occurs.
5. On a reserved operand fault:
   - If PSL⟨FPD⟩ = 0, the reserved operand is the degree operand (greater than 31), the argument operand, or some coefficient.
   - If PSL⟨FPD⟩ = 1, the reserved operand is a coefficient, and R3 (except for POLYH) or R5 (for POLYH) is pointing at the value that caused the exception.
   - The state of the saved condition codes and the other registers is UNPREDICTABLE. If the fault handler changes the reserved operand and preserves the contents of the condition codes and all registers, the fault is continuable.
6. On floating underflow after the rounding operation at any iteration of the computation loop, a fault occurs if PSL⟨FU⟩ is set. If PSL⟨FU⟩ is clear, the temporary result (tmp3) is replaced by 0

and the operation continues. In this case, the final result may be non-zero if underflow occurred before the last iteration.

7. On floating overflow after the rounding operation at any iteration of the computation loop, the instruction terminates with a fault.
8. If the argument is zero, the result is C[0]. Additionally, if one of the coefficients in the table (other than C[0]) is a reserved operand, whether a reserved operand fault occurs is UNPREDICTABLE.
9. The POLYx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11.
10. The POLYx instructions are in the emulated-only group and are unlikely to be implemented in future VAX processors. Software developers are advised to avoid using these instructions.
11. For POLYH, some implementations may not save arg on the stack until after an interrupt or fault occurs. However, arg will always be on the stack if an interrupt or floating fault occurs after FPD is set. If the four longwords on the stack overlap any of the source operands, the results are UNPREDICTABLE.

SUB Subtract

Format

```
opcode sub.rx, dif.mx                2 operand
opcode sub.rx, min.rx, dif.wx        3 operand
```

Operation

```
dif ← dif - sub;                     !2 operand
dif ← min - sub;                     !3 operand
```

Condition codes

```
N ← dif LSS 0;
Z ← dif EQL 0;
V ← 0;
C ← 0;
```

Exceptions

```
floating overflow
floating underflow
reserved operand
```

Opcodes

42	SUBF2	Subtract F_floating 2 Operand
43	SUBF3	Subtract F_floating 3 Operand
62	SUBD2	Subtract D_floating 2 Operand
63	SUBD3	Subtract D_floating 3 Operand
42FD	SUBG2	Subtract G_floating 2 Operand
43FD	SUBG3	Subtract G_floating 3 Operand
62FD	SUBH2	Subtract H_floating 2 Operand
63FD	SUBH3	Subtract H_floating 3 Operand

Description

In 2 operand format, the subtrahend operand is subtracted from the difference operand and the difference operand is replaced by the rounded result. In 3 operand format, the subtrahend operand is subtracted from the minuend operand and the difference operand is replaced by the rounded result.

Notes

1. On a reserved operand fault, the difference operand is unaffected and the condition codes are UNPREDICTABLE.
2. On floating underflow, a fault occurs if PSL⟨FU⟩ is set. Zero is stored as the result of floating underflow only if PSL⟨FU⟩ is clear. On a floating underflow fault, the difference operand is unaffected. If PSL⟨FU⟩ is clear, the difference operand is replaced by 0 and no exception occurs.
3. On floating overflow, the instruction faults. The difference operand is unaffected, and the condition codes are UNPREDICTABLE.
4. The SUBHx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

TST Test

Format

```
opcode src.rx
```

Operation

```
src - 0;
```

Condition codes

```
N ← src LSS 0;
Z ← src EQL 0;
V ← 0;
C ← 0;
```

Exceptions

```
reserved operand
```

Opcodes

53	TSTF	Test F_floating
73	TSTD	Test D_floating
53FD	TSTG	Test G_floating
73FD	TSTH	Test H_floating

Description

The condition codes are affected according to the value of the source operand.

Notes

1. *TSTx src* is equivalent to *CMPx src, #0*, but is 5 bytes (F_floating), 9 bytes (D_floating or G_floating), or 17 bytes (H_floating) shorter.
2. On a reserved operand fault, the condition codes are UNPREDICTABLE.
3. The TSTH instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

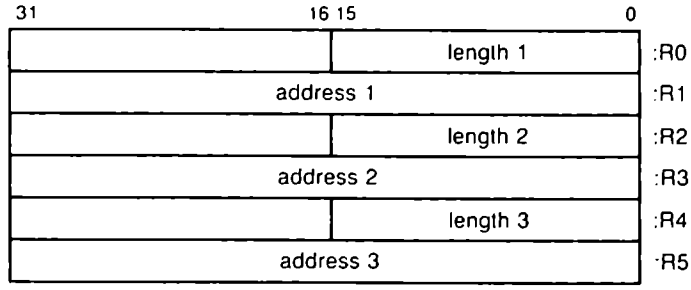


Figure 3.7
Character-String-Instruction Control Block

3.11 *Character-String Instructions*

A character string is specified by two operands:

- An unsigned-word operand that specifies the length of the character string in bytes
- The address of the lowest addressed byte of the character string, which is specified by a byte operand of address access type

Each character-string instruction uses general registers R0 and R1, R0 through R3, or R0 through R5 to contain a control block that maintains updated addresses and state during the execution of the instruction. When instruction execution is completed, these registers are available to software to use as string specification operands for a subsequent instruction on a contiguous character string. During the execution of the instructions, pending interrupt conditions are tested. If any is found, the control block is updated, a first-part-done bit is set in the PSL, and the instruction interrupted (see Chapter 5). After the interruption, the instruction resumes transparently.

The format of the control block is shown in Figure 3.7. The fields length 1, length 2 (if required), and length 3 (if required) contain the number of bytes remaining to be processed in the first, second, and third string operands, respectively. The fields address 1, address 2 (if required), and address 3 (if required) contain the address of the next byte to be processed in the first, second, and third string operands, respectively.

Memory access faults will not occur when a zero-length string is specified, because no memory reference occurs.

The MATCHC, MOVTC, and MOVTUC instructions are optional in implementations of the VAX architecture. Execution of an omitted instruction results in an emulated instruction exception. Omitted instructions are emulated by Digital-supported operating systems. Emulation software may allocate and use current-mode stack space when executed. For more detail, refer to Chapter 11.

CMPC Compare Characters

Format

```
opcode len.rw, src1addr.ab, src2addr.ab,   3 operand
       {R0-3.wl}
opcode src1len.rw, src1addr.ab, fill.rb,
       src2len.rw, src2addr.ab, {R0-3.wl}  5 operand
.end literal
.literal
```

Operation

```
! In all cases and flows, tmpa and tmpb hold the last
! two bytes that were compared and so are used to set
! the condition codes at the end.

CMPC3 Flow:

tmp1 ← len;        tmp2 ← src1addr;  !3 operand
tmp4 ← src2addr;
tmpa ← 0;          tmpb ← 0;         !init compare regs

while {tmp1 NEQU 0} do               !SRC1 & SRC2
   begin                             !not zero-length
   tmpa ← (tmp2);  tmpb ← (tmp4);
   if tmpa EQL tmpb then             !bytes match
      begin
      tmp1 ← tmp1 - 1; tmp2 ← tmp2 + 1;
      tmp4 ← tmp4 + 1;
      end
   else [exit while loop];           !mismatch
   end;

tmp3 ← tmp1;
[exit to Register Save flow];

CMPC5 Flow:

tmp1 ← src1len;  tmp2 ← src1addr;   !5 operand
```

```
tmp3 ← src2len;  tmp4 ← src2addr;
tmpa ← 0;        tmpb ← 0;                !init compare regs

while {tmp1 NEQU 0} AND {tmp3 NEQU 0} do  !SRC1 & SRC2
   begin                                  !not zero-length
   tmpa ← (tmp2);  tmpb ← (tmp4);
   if tmpa EQL tmpb then                  !bytes match
      begin
      tmp1 ← tmp1 - 1;  tmp2 ← tmp2 + 1;
      tmp3 ← tmp3 - 1;  tmp4 ← tmp4 + 1;
      end
   else [exit while loop];                !mismatch
   end;

while ({tmp1 NEQU 0} AND {tmp3 EQU 0}) do !SRC2 zero-length
   begin                                  !or exhausted, SRC1
   tmpb ← fill;                           !has bytes left
   tmpa ← (tmp2);
   if tmpa EQL fill then                  !bytes match
      begin
      tmp1 ← tmp1 - 1;  tmp2 ← tmp2 + 1;
      end
   else [exit while loop];                !mismatch
   end;

while ({tmp3 NEQU 0} AND {tmp1 EQU 0}) do !SRC1 zero-length
   begin                                  !or exhausted, SRC2
   tmpa ← fill;                           !has bytes left
   tmpb ← (tmp4);
   if fill EQL tmpb then                  !bytes match
      begin
      tmp3 ← tmp3 - 1;  tmp4 ← tmp4 + 1;
      end
   else [exit while loop];                !mismatch
   end;

Register Save Flow:

r0 ← tmp1;
r1 ← tmp2;
r3 ← tmp3;
r4 ← tmp4;
```

Condition codes

```
N ← tmpa LSS tmpb;
Z ← tmpa EQL tmpb;
V ← 0;
C ← tmpa LSSU tmpb;
```

Exceptions

none

Opcodes

29	CMPC3	Compare Characters 3 Operand
2D	CMPC5	Compare Characters 5 Operand

Description

In 3 operand format, the bytes of string 1 specified by the length and address 1 operands are compared with the bytes of string 2 specified by the length and address 2 operands. Comparison proceeds until inequality is detected or all the bytes of the strings have been examined. Condition codes are affected by the result of the last byte comparison. In 5 operand format, the bytes of the string 1 specified by the length 1 and address 1 operands are compared with the bytes of the string 2 specified by the length 2 and address 2 operands. If one string is longer than the other, the shorter string is conceptually extended to the length of the longer by appending (at higher addresses) bytes equal to the fill operand. Comparison proceeds until inequality is detected or all the bytes of the strings have been examined. Condition codes are affected by the result of the last byte comparison. For either CMPC3 or CMPC5, two zero-length strings compare equal (PSL⟨Z⟩ is set and PSL⟨N⟩, PSL⟨V⟩, and PSL⟨C⟩ are cleared.)

Notes

1. After execution of CMPC3:
   - R0 = number of bytes remaining in string 1 (including byte that terminated comparison); R0 is zero only if strings are equal
   - R1 = address of the byte in string 1 that terminated comparison; if strings are equal, address of one byte beyond string 1
   - R2 = R0
   - R3 = address of the byte in string 2 that terminated comparison; if strings are equal, address of one byte beyond string 2

2. After execution of CMPC5:
   - R0 = number of bytes remaining in string 1 (including byte that terminated comparison); R0 is zero only if string 1 and string 2 are of equal length and equal or string 1 was exhausted before comparison terminated
   - R1 = address of the byte in string 1 that terminated comparison; if comparison did not terminate before string 1 exhausted, address of one byte beyond string 1
   - R2 = number of bytes remaining in string 2 (including byte that terminated comparison); R2 is zero only if string 2 and string 1 are of equal length or string 2 was exhausted before comparison terminated

- R3 = address of the byte in string 2 that terminated comparison; if comparison did not terminate before string 2 was exhausted, address of one byte beyond string 2

3. If both strings have zero length, PSL⟨Z⟩ is set and PSL⟨N⟩, PSL⟨V⟩, and PSL⟨C⟩ are cleared just as in the case of two equal strings.

LOCC Locate Character

Format

```
opcode char.rb, len.rw, addr.ab, {R0-1.wl}
```

Operation

```
tmp1 ← len;
tmp2 ← addr;
if tmp1 GTRU 0 then
   begin
   while {tmp1 NEQ 0} AND {(tmp2) NEQ char}  do
      begin
      tmp1 ← tmp1 - 1;
      tmp2 ← tmp2 + 1;
      end;
   end;
R0 ← tmp1;
R1 ← tmp2;
```

Condition codes

```
N ← 0;
Z ← R0 EQL 0;
V ← 0;
C ← 0;
```

Exceptions

none

Opcodes

3A LOCC Locate Character

Description

The character operand is compared with the bytes of the string specified by the length and address operands. Comparison continues until equality is detected or all bytes of the string have been compared. If equality is detected, the condition code Z-bit is cleared; otherwise, the Z-bit is set.

Notes

1. After execution:
   - R0 = number of bytes remaining in the string (including located one) if byte located; otherwise, 0
   - R1 = address of the byte located if byte located; otherwise, address of one byte beyond the string
2. If the string has zero length, condition code Z is set just as though each byte of the entire string were unequal to character.

MATCHC Match Characters

Format

```
opcode objlen.rw, objaddr.ab, srclen.rw, srcaddr.ab,
       {R0-3.wl}
```

Operation

```
tmp1 ← objlen;
tmp2 ← objaddr;
tmp3 ← srclen;
tmp4 ← srcaddr;
tmp5 ← tmp1;

while {tmp1 NEQU 0} AND {tmp3 GEQU tmp1} do
   begin
   if (tmp2) EQL (tmp4) then
      begin
      tmp1 ← tmp1 - 1;
      tmp2 ← tmp2 + 1;
      tmp3 ← tmp3 - 1;
      tmp4 ← tmp4 + 1;
      end
   else
      begin
      tmp2 ← tmp2 - ZEXT (tmp5-tmp1);
      tmp3 ← {tmp3 - 1} + {tmp5-tmp1};
      tmp4 ← {tmp4 + 1} - ZEXT (tmp5-tmp1);
      tmp1 ← tmp5;
      end;
   end;

if {tmp3 LSSU tmp1} then
   begin
   tmp4 ← tmp4 + tmp3;
   tmp3 ← 0;
   end;

R0 ← tmp1;
R1 ← tmp2;
R2 ← tmp3;
R3 ← tmp4;
```

Condition codes

```
N ← 0;
Z ← R0 EQL 0;        !match found
V ← 0;
C ← 0;
```

Exceptions

none

Opcodes

39 MATCHC Match Characters

Description

The source string specified by the source length and source address operands is searched for a substring that matches the object string specified by the object length and object address operands. If the substring is found, the condition code Z-bit is set; otherwise, it is cleared.

Notes

1. After execution:
   - R0 = if a match occurred, 0; otherwise, the number of bytes in the object string
   - R1 = if a match occurred, the address of one byte beyond the object string (that is, objaddr + objlen); otherwise, the address of the object string
   - R2 = if a match occurred, the number of bytes remaining in the source string; otherwise, 0
   - R3 = if a match occurred, the address of one byte beyond the last byte matched; otherwise, the address of one byte beyond the source string (that is, srcaddr + srclen)

   For zero-length source and object strings, R3 and R1 contain the source and object addresses, respectively.
2. If both strings have zero length or if the object string has zero length, PSL⟨Z⟩is set and registers R0 through R3 are left just as though the substring were found.
3. If the source string has zero length and the object string has non-zero length, PSL⟨Z⟩ is cleared and registers R0 through R3 are left just as though the substring were not found.
4. The MATCHC instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.
5. The MATCHC instruction is in the emulated-only group and is unlikely to be implemented in future VAX processors. Software developers are advised to avoid using this instruction.

MOVC Move Character

Format

```
opcode len.rw, srcaddr.ab, dstaddr.ab,      3 operand
       {R0-5.wl}
opcode srclen.rw, srcaddr.ab, fill.rb,      5 operand
       dstlen.rw, dstaddr.ab, {R0-5.wl}
```

Operation

```
tmp1 ← len;                                 !3 operand
tmp2 ← srcaddr;
tmp3 ← dstaddr;
if tmp2 GTRU tmp3 then
   begin
   while tmp1 NEQU 0 do
      begin
      (tmp3) ← (tmp2);
      tmp1 ← tmp1 - 1;
      tmp2 ← tmp2 + 1;
      tmp3 ← tmp3 + 1;
      end;
   R1 ← tmp2;
   R3 ← tmp3;
   end
else
   begin
   tmp4 ← tmp1;
   tmp2 ← tmp2 + ZEXT(tmp1);
   tmp3 ← tmp3 + ZEXT(tmp1);
   while tmp1 NEQU 0 do
      begin
      tmp1 ← tmp1 - 1;
      tmp2 ← tmp2 - 1;
      tmp3 ← tmp3 - 1;
      (tmp3) ← (tmp2);
      end;
   R1 ← tmp2 + ZEXT(tmp4);
   R3 ← tmp3 + ZEXT(tmp4);
   end;
R0 ← 0;
R2 ← 0;
R4 ← 0;
R5 ← 0;

tmp1 ← srclen;                              !5 operand
tmp2 ← srcaddr;
tmp3 ← dstlen;
tmp4 ← dstaddr;
if tmp2 GTRU tmp4 then
   begin
```

MOVC

```
while {tmp1 NEQU 0} AND {tmp3 NEQU 0} do
   begin
   (tmp4) ← (tmp2);
   tmp1 ← tmp1 - 1;
   tmp2 ← tmp2 + 1;
   tmp3 ← tmp3 - 1;
   tmp4 ← tmp4 + 1;
   end;
while tmp3 NEQU 0 do
   begin
   (tmp4) ← fill;
   tmp3 ← tmp3 - 1;
   tmp4 ← tmp4 + 1;
   end;
R1 ← tmp2;
R3 ← tmp4;
end
else
   begin
   tmp5 ← MINU(tmp1, tmp3);
   tmp6 ← tmp3;
   tmp2 ← tmp2 + ZEXT(tmp5);
   tmp4 ← tmp4 + ZEXT(tmp6);
   while tmp3 GTRU tmp1 do
      begin
      tmp3 ← tmp3 - 1;
      tmp4 ← tmp4 - 1;
      (tmp4) ← fill;
      end;
   while tmp3 NEQU  0 do
      begin
      tmp1 ← tmp1 - 1;
      tmp2 ← tmp2 - 1;
      tmp3 ← tmp3 - 1;
      tmp4 ← tmp4 - 1;
      (tmp4) ← (tmp2);
      end;
   R1 ← tmp2 + ZEXT (tmp5);
   R3 ← tmp4 + ZEXT (tmp6);
   end;
R0 ← tmp1;
R2 ← 0;
R4 ← 0;
R5 ← 0;
```

Condition codes

```
N ← 0;                    !MOVC3
Z ← 1;
V ← 0;
C ← 0;
```

```
N ← srclen LSS dstlen;     !MOVC5
Z ← srclen EQL dstlen;
V ← 0;
C ← srclen LSSU dstlen;
```

Exceptions none

Opcodes

28	MOVC3	Move Character 3 Operand
2C	MOVC5	Move Character 5 Operand

Description In 3 operand format, the destination string specified by the length and destination address operands is replaced by the source string specified by the length and source address operands. In 5 operand format, the destination string specified by the destination length and destination address operands is replaced by the source string specified by the source length and source address operands. If the destination string is longer than the source string, the highest addressed bytes of the destination are replaced by the fill operand. If the destination string is shorter than the source string, the highest addressed bytes of the source string are not moved. The operation of the instruction is such that overlap of the source and destination strings does not affect the result.

Notes

1. After execution of MOVC3:
   - R0 = 0
   - R1 = address of one byte beyond the source string
   - R2 = 0
   - R3 = address of one byte beyond the destination string
   - R4 = 0
   - R5 = 0
2. After execution of MOVC5:
   - R0 = number of unmoved bytes remaining in source string; R0 is non-zero only if source string is longer than destination string
   - R1 = address of one byte beyond the last byte in source string that was moved
   - R2 = 0
   - R3 = address of one byte beyond the destination string
   - R4 = 0
   - R5 = 0
3. MOVC3 is the preferred way to copy one block of memory to another.
4. MOVC5 with a zero source length operand is the preferred way to fill a block of memory with the fill character.

MOVTC Move Translated Characters

Format

```
opcode srclen.rw, srcaddr.ab, fill.rb, tbladdr.ab,
       dstlen.rw, dstaddr.ab, {R0-5.wl}
```

Operation

```
tmp1 ← srclen;  tmp2 ← srcaddr;
tmp3 ← dstlen;  tmp4 ← dstaddr;
if tmp2 GTRU tmp4 then
   begin
   while {tmp1 NEQU 0} AND {tmp3 NEQU 0}
      begin
      (tmp4) ← (tbladdr + ZEXT((tmp2)));
      tmp1 ← tmp1 - 1;
      tmp2 ← tmp2 + 1;
      tmp3 ← tmp3 - 1;
      tmp4 ← tmp4 + 1;
      end;
   while {tmp3 NEQU 0} do
      begin
      (tmp4) ← fill;
      tmp3 ← tmp3 - 1;
      tmp4 ← tmp4 + 1;
      end;
   R1 ← tmp2;
   R5 ← tmp4;
   end;
else
   begin
   tmp5 ← MINU(tmp1,tmp3);
   tmp6 ← tmp3;
   tmp2 ← tmp2 + ZEXT(tmp5);
   tmp4 ← tmp4 + ZEXT(tmp6);
   while tmp3 GTRU tmp1 do
      begin
      tmp3 ← tmp3 - 1;
      tmp4 ← tmp4 - 1;
      (tmp4) ← fill;
      end;
   while tmp3 NEQU 0 do
      begin
      tmp1 ← tmp1 - 1;
      tmp2 ← tmp2 - 1;
      tmp3 ← tmp3 - 1;
      tmp4 ← tmp4 - 1;
      (tmp4) ← (tbladdr + ZEXT((tmp2)));
      end;
   R1 ← tmp2 + ZEXT(tmp5);
   R5 ← tmp4 + ZEXT(tmp6);
```

```
    end;
R0 ← tmp1;      R2 ← 0;
R3 ← tbladdr;   R4 ← 0;
```

Condition codes

```
N ← srclen LSS dstlen;
Z ← srclen EQL dstlen;
V ← 0;
C ← srclen LSSU dstlen;
```

Exceptions

none

Opcodes

2E MOVTC Move Translated Characters

Description

The source string specified by the source length and source address operands is translated and replaces the destination string specified by the destination length and destination address operands. Translation is accomplished by using each byte of the source string as an index into a 256-byte table whose zeroth entry address is specified by the table address operand. The byte selected replaces the byte of the destination string. If the destination string is longer than the source string, the highest addressed bytes of the destination string are replaced by the fill operand. If the destination string is shorter than the source string, the highest addressed bytes of the source string are not translated and moved. The operation of the instruction is such that overlap of the source and destination strings does not affect the result. If the destination string overlaps the translation table, the destination string is UNPREDICTABLE.

Notes

1. After execution:
   - R0 = number of untranslated bytes remaining in source string; R0 is non-zero only if source string is longer than destination string
   - R1 = address of one byte beyond the last byte in source string that was translated
   - R2 = 0
   - R3 = address of the translation table
   - R4 = 0
   - R5 = address of one byte beyond the destination string
2. The MOVTC instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.
3. The MOVTC instruction is in the emulated-only group and is unlikely to be implemented in future VAX processors. Software developers are advised to avoid using this instruction.

MOVTUC Move Translated Until Character

Format

```
opcode srclen.rw, srcaddr.ab, esc.rb, tbladdr.ab, dstlen.rw,
       dstaddr.ab, {R0-5.wl}
```

Operation

```
tmp1 ← srclen;
tmp2 ← srcaddr;
tmp3 ← dstlen;
tmp4 ← dstaddr;

if tmp1 GTRU 0 and tmp3 GTRU 0 then
   begin
   while {tmp1 NEQU 0} AND {tmp3 NEQU 0} do
      if{(tbladdr + ZEXT(tmp2)) NEQU esc} then
         begin
         (tmp4) ← (tbladdr + ZEXT(tmp2));
         tmp1 ← tmp1 - 1;
         tmp2 ← tmp2 + 1;
         tmp3 ← tmp3 - 1;
         tmp4 ← tmp4 + 1;
         end;

      else exit while loop;
   end;

R0 ← tmp1;
R1 ← tmp2;
R2 ← 0;
R3 ← tbladdr;
R4 ← tmp3;
R5 ← tmp4;
```

Condition codes

```
N ← srclen LSS dstlen;
Z ← srclen EQL dstlen;
V ← {terminated by escape};
C ← srclen LSSU dstlen;
```

Exceptions

none

Opcodes

2F MOVTUC Move Translated Until Character

Description

The source string specified by the source length and source address operands is translated and replaces the destination string specified by the destination length and destination address operands. Translation is accomplished by using each byte of the source string as index into a 256-byte table whose zeroth entry address is specified by the table address operand. The byte selected replaces the byte of the destination string. Translation continues until a

translated byte is equal to the escape byte or until the source string or destination string is exhausted. If translation is terminated because of escape, the condition code V-bit is set; otherwise, it is cleared. If the destination string overlaps the table, the destination string and registers R0 through R5 are UNPREDICTABLE. If the source and destination strings overlap and their addresses are not identical, the destination string and registers R0 through R5 are UNPREDICTABLE. If the source and destination string addresses are identical, the translation is performed correctly.

1. After execution:
   - R0 = number of bytes remaining in source string (including the byte that caused the escape); R0 is zero only if the entire source string was translated and moved without escape
   - R1 = address of the byte that resulted in destination string exhaustion or escape; or if no exhaustion or escape, address of one byte beyond the source string
   - R2 = 0
   - R3 = address of the table
   - R4 = number of bytes remaining in the destination string
   - R5 = address of the byte in the destination string that would have received the translated byte that caused the escape or would have received a translated byte if the source string were not exhausted; or if no exhaustion or escape, the address of one byte beyond the destination string

2. The MOVTUC instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.
3. The MOVTUC instruction is in the emulated-only group and is unlikely to be implemented in future VAX processors. Software developers are advised to avoid using this instruction.

SCANC Scan Characters

Format

```
opcode len.rw, addr.ab, tbladdr.ab, mask.rb, {R0-3.wl}
```

Operation

```
tmp1 ← len;
tmp2 ← addr;
if tmp1 GTRU 0 then
   while {tmp1 NEQU 0} AND
         {{(tbladdr + ZEXT((tmp2))) AND mask} EQL 0} do
      begin
      tmp1 ← tmp1 - 1;
      tmp2 ← tmp2 + 1;
      end;
R0 ← tmp1;
R1 ← tmp2;
R2 ← 0;
R3 ← tbladdr;
```

Condition codes

```
N ← 0;
Z ← R0 EQL 0;
V ← 0;
C ← 0;
```

Exceptions

none

Opcodes

2A SCANC Scan Characters

Description

The bytes of the string specified by the length and address operands are successively used to index into a 256-byte table whose zeroth entry address is specified by the table address operand. The byte selected from the table is ANDed with the mask operand. The operation continues until the result of the AND is non-zero or all the bytes of the string have been exhausted. If a non-zero AND result is detected, the condition code Z-bit is cleared; otherwise, the Z-bit is set.

Notes

1. After execution:
   - R0 = number of bytes remaining in the string (including the byte that produced the non-zero AND result); R0 is zero only if there was no non-zero AND result
   - R1 = address of byte that produced non-zero AND result; or, if no such byte, address of one byte beyond the string
   - R2 = 0
   - R3 = address of the table
2. If the string has zero length, PSL⟨Z⟩ is set just as though the entire string were scanned.

SKPC Skip Character

Format

```
opcode char.rb, len.rw, addr.ab, {R0-1.wl}
```

Operation

```
tmp1 ← len;
tmp2 ← addr;
if tmp1 GTRU 0 then
   begin
   while {tmp1 NEQ 0} AND {(tmp2) EQL char} do
      begin
      tmp1 ← tmp1 - 1;
      tmp2 ← tmp2 + 1;
      end;
   end;
R0 ← tmp1;
R1 ← tmp2;
```

Condition codes

```
N ← 0;
Z ← R0 EQL 0;
V ← 0;
C ← 0;
```

Exceptions

none

Opcodes

3B SKPC Skip Character

Description

The character operand is compared with the bytes of the string specified by the length and address operands. Comparison continues until inequality is detected or all bytes of the string have been compared. If inequality is detected, the condition code Z-bit is cleared; otherwise, the Z-bit is set.

Notes

1. After execution:
   - R0 = number of bytes remaining in the string (including the unequal one) if unequal byte located; otherwise, 0
   - R1 = address of the byte located if byte located; otherwise, address of one byte beyond the string
2. If the string has zero length, PSL⟨Z⟩ is set just as though each byte of the entire string were equal to character.

SPANC Span Characters

Format

```
opcode len.rw, addr.ab, tbladdr.ab, mask.rb, {R0-3.wl}
```

Operation

```
tmp1 ← len;
tmp2 ← addr;
if tmp1 GTRU 0 then
   while {tmp1 NEQU 0} AND
         {{(tbladdr + ZEXT((tmp2))) AND mask} NEQ 0} do
      begin
      tmp1 ← tmp1 - 1;
      tmp2 ← tmp2 + 1;
      end;
R0 ← tmp1;
R1 ← tmp2;
R2 ← 0;
R3 ← tbladdr;
```

Condition codes

```
N ← 0;
Z ← R0 EQL 0;
V ← 0;
C ← 0;
```

Exceptions

none

Opcodes

2B SPANC Span Characters

Description

The bytes of the string specified by the length and address operands are successively used to index into a 256-byte table whose zeroth entry address is specified by the table address operand. The byte selected from the table is ANDed with the mask operand. The operation continues until the result of the AND is zero or all the bytes of the string have been exhausted. If a zero AND result is detected, the condition code Z-bit is cleared; otherwise, the Z-bit is set.

Notes

1. After execution:
   - R0 = number of bytes remaining in the string (including the byte that produced the zero AND result); R0 is zero only if there was no zero AND result
   - R1 = address of byte that produced a zero AND result; or, if no such byte, address of one byte beyond the string
   - R2 = 0
   - R3 = address of the table
2. If the string has zero length, the PSL⟨Z⟩ is set just as though the entire string were spanned.

3.12 *Cyclic Redundancy Check Instruction*

This instruction is designed to implement the calculation and checking of a cyclic redundancy check (CRC) for any CRC polynomial up to 32 bits. Cyclic redundancy checking is an error-detection method involving a division of the data stream by a CRC polynomial. The data stream is represented as a standard VAX string in memory. Error detection is accomplished by computing the CRC at the source and again at the destination, and comparing the CRC computed at each end. The choice of the polynomial is such as to minimize the number of undetected block errors of specific lengths. The choice of a CRC polynomial is not given here; refer to "Cyclic Codes for Error Detection," by W. Peterson and D. Brown in the *Proceedings of the IRE* (January 1961).

The operands to the CRC instruction are a string descriptor, a 16-longword table, and an initial CRC. The string descriptor is a standard VAX operand pair of the length of the string in bytes (up to 65,535) and the starting address of the string. The contents of the table are a function of the CRC polynomial to be used. The table can be calculated from the polynomial by the algorithm in the notes. Several common CRC polynomials are also included in the notes. The initial CRC is used to start the polynomial correctly. Typically, it has the value 0 or −1, but would be different if the data stream were represented by a sequence of non-contiguous strings.

The CRC instruction operates by scanning the string and, for each byte of the data stream, including it in the CRC being calculated. The byte is included by XORing it to the right 8 bits of the CRC. Then the CRC is shifted right 1 bit, inserting zero on the left. The right-most bit of the CRC (lost by the shift) is used to control the XORing of the CRC polynomial with the resultant CRC. If the bit is set, the polynomial is XORed with the CRC. Then the CRC is again shifted right, and the polynomial is conditionally XORed with the result a total of eight times. The actual algorithm used can shift by 1, 2, or 4 bits at a time using the appropriate entries in a specially constructed table. The instruction produces a 32-bit CRC. For shorter polynomials, the result must be extracted from the 32-bit field. The data stream must be a multiple of 8 bits in length. If it is not, the stream must be right-adjusted in the string with leading 0 bits.

The cyclic-redundancy-check instruction is optional in implementations of the VAX architecture. Execution of the instruction, when omitted, results in an emulated instruction exception. Omitted instructions are emulated by Digital-supported operating systems. Emulation software may allocate and use current-mode stack space when executed. For more detail, refer to Chapter 11.

CRC — Calculate Cyclic Redundancy Check

Format

```
opcode tbl.ab, inicrc.rl, strlen.rw, stream.ab, {R0-3.wl}
```

Operation

```
tmp1 ← strlen;
tmp2 ← stream;
tmp3 ← inicrc;
tmp4 ← tbl;
while tmp1 NEQU 0 do
   begin
   tmp3<7:0>← tmp3<7:0> XOR (tmp2)+;
   for tmp5 ← 1,limit do        !see notes for limit,s,i
      tmp3 ← ZEXT(tmp3<31:s>) XOR
              (tmp4 + {4*ZEXT(tmp3<s-1:0>*i)};
   tmp1 ← tmp1 -1;
   end;
R0 ← tmp3;
R1 ← 0;
R2 ← 0;
R3 ← tmp2;
```

Condition codes

```
N ← R0 LSS 0;
Z ← R0 EQL 0;
V ← 0;
C ← 0;
```

Exceptions

none

Opcodes

0B CRC Calculate Cyclic Redundancy Check

Description

The CRC of the data stream described by the string descriptor is calculated. The initial CRC is given by inicrc and is normally 0 or −1 unless the CRC is calculated in several steps. The result is left in R0. If the polynomial is less than order 32, the result must be extracted from the result. The CRC polynomial is expressed by the contents of the 16-longword table. See the notes for the calculation of the table.

1. If the data stream is not a multiple of 8-bits long, it must be right-adjusted with leading 0 fill.
2. If the CRC polynomial is less than order 32, the result must be extracted from the low-order bits of R0.
3. The following algorithm can be used to calculate the CRC table given a polynomial expressed as follows:

   ```
   polyn<n> ← {coefficient of x**(order -1-n)}
   ```

   This routine is available as the VAX/VMS system library routine LIB$CRC_TABLE (poly.rl, table.ab). The bits of the poly operand, taken right to left, represent the coefficients of the polynomial, taken left to right and skipping the most significant bit. The table is the location of a 64-byte (16-longword) table into which the result will be written.

   ```
         SUBROUTINE LIB$CRC_TABLE (POLY, TABLE)
         INTEGER*4 POLY, TABLE(0:15), TMP, X
         DO 190 INDEX = 0, 15
         TMP = INDEX
         DO 150 I = 1, 4
         X = TMP .AND. 1
         TMP = ISHFT(TMP,-1)     !logical shift right one bit
         IF (X .EQ. 1) TMP =  TMP .XOR. POLY
   150   CONTINUE
         TABLE(INDEX) = TMP
   190   CONTINUE
         RETURN
         END
   ```

4. Table 3.1 describes some commonly used CRC polynomials.
5. This instruction produces an UNPREDICTABLE result unless the table is well formed, such as produced in note 3, above. Note that for any well formed table, entry[0] is always 0 and entry[8] is always the polynomial expressed, as in note 3. The operation can be implemented using shifts of 1, 2, or 4 bits at a time, as shown in Table 3.2.
6. If the stream has zero length, R0 receives the initial CRC.
7. The CRC instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.
8. The CRC instruction is in the emulated-only group and is unlikely to be implemented in future VAX processors. Software developers are advised to avoid using this instruction.

Table 3.1
Common CRC Polynomials

Polynomial	POLY	Initial Value	Result
CRC-16 (used for DDCMP and Bisync)			
$x^{16} + x^{15} + x^2 + 1$	0000A001	00000000	R0⟨15:0⟩
CCITT (used for ADCCP, HDLC, SDLC)			
$x^{16} + x^{12} + x^5 + 1$	00008408	0000FFFF	one's complement of R0⟨15:0⟩
AUTODIN-II			
$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$	EDB88320	FFFFFFFF	one's complement of R0⟨31:0⟩

Table 3.2
CRC Shift Amounts

Shift Amount(s)	Steps Per Byte (Limit)	Table Index	Table Index Multiplier (i)	Table Entries Used
1	8	tmp3⟨0⟩	8	[0]=0,[8]
2	4	tmp3⟨1:0⟩	4	[0]=0,[4],[8],[12]
4	2	tmp3⟨3:0⟩	1	all

3.13 *Decimal-String Instructions*

Decimal-string instructions operate on packed decimal strings. Convert instructions are provided between packed decimal and trailing numeric string (overpunched and zoned) and leading separate numeric string formats. Where necessary, a specific data type is identified. Where the phrase decimal string is used, it means any of the three data types.

A decimal string is specified by two operands:

- The first operand is the length, the number of digits in the string. The number of bytes in the string is a function of the length and the type of decimal string referenced (see Chapter 1).
- The second operand is the address of the lowest addressed byte of the string. This byte contains the most significant digit for trailing numeric and packed decimal strings. This byte contains a sign for left separate numeric strings. The address is specified by a byte operand of address access type.

Each of the decimal-string instructions uses general registers R0 through R3 or R0 through R5 to contain a control block that maintains updated addresses and state during the execution of the instruction. At completion, the registers containing addresses are available to the software to use as string specification operands for a subsequent instruction on the same decimal strings. During the execution of the instructions, pending interrupt conditions are tested; if any is found, the control block is updated. First-part-done is set in the PSL, and the instruction interrupted (see Chapter 5). After the interruption, the instruction resumes transparently. The format of the control block at completion is shown in Figure 3.8. The fields address 1, address 2, and address 3 (if required) contain the address of the byte containing the most significant digit of the first, second, and third (if required) string operands, respectively.

The decimal-string instructions treat decimal strings as integers with the decimal point assumed immediately beyond the least significant digit of the string. If a string in which a result is to be stored is longer than the result, its most significant digits are filled with zeros.

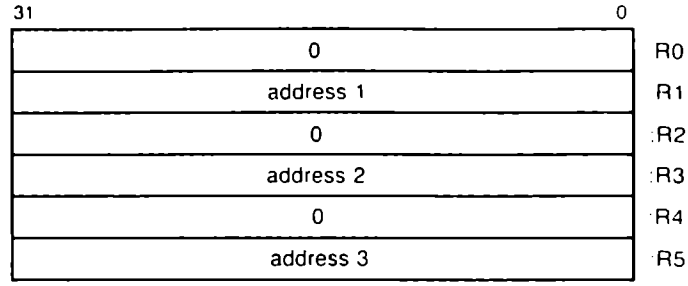


Figure 3.8
Decimal-String-Instruction Control Block

Decimal-string instructions are optional in implementations of the VAX architecture. Execution of an omitted instruction results in an emulated instruction exception. Omitted instructions are emulated by Digital-supported operating systems. Emulation software may allocate and use current-mode stack space when executed. For more detail, refer to Chapter 11.

3.13.1 *Decimal Overflow*

Decimal overflow occurs if the destination string is too short to contain all the digits (excluding leading zeros) of the result. On overflow, the destination string is replaced by the correctly signed least significant digits of the true result (even if the stored result is –0). Note that neither the high nibble of an even-length packed decimal string nor the sign byte of a leading separate numeric string is used to store result digits.

3.13.2 *Zero Numbers*

A zero result has a positive sign for all operations that complete without decimal overflow, except for CVTPT, which does not fix a –0 to a +0. When digits are lost because of overflow, however, a zero result receives the sign (positive or negative) of the correct result.

A decimal string with value –0 is treated as identical to a decimal string with value +0. For example, +0 compares equal to –0. When condition codes are affected on a –0 result, they are affected as if the result were +0; that is, PSL⟨N⟩ is cleared and PSL⟨Z⟩ is set.

3.13.3 *Reserved Operand Exception*

A reserved operand abort occurs if the length of a decimal-string operand is outside the range 0 through 31 or if an invalid sign or

digit is encountered in CVTSP and CVTTP. The PC points to the opcode of the instruction causing the exception.

3.13.4 *UNPREDICTABLE Results*

The result of any operation is UNPREDICTABLE if any source decimal-string operand contains invalid data. Except for CVTSP and CVTTP, the decimal-string instructions do not verify the validity of source operand data.

If the destination operands overlap any source operands, the result of an operation will, in general, be UNPREDICTABLE. The destination strings, registers used by the instruction, and condition codes, will in general, be UNPREDICTABLE when a reserved operand abort occurs.

3.13.5 *Packed Decimal Operations*

Packed decimal strings generated by the decimal-string instructions always have the preferred sign representation: 12 for plus and 13 for minus. An even-length packed decimal string is always generated with a "0" digit in the high nibble of the first byte of the string.

A packed decimal string contains an invalid nibble if:

1. A digit occurs in the sign position
2. A sign occurs in a digit position
3. For an even-length string, a non-zero nibble occurs in the high order nibble of the lowest addressed byte

3.13.6 *Zero-Length Decimal Strings*

The length of a packed decimal string can be 0. In this case, the value is zero (plus or minus) and one byte of storage is occupied. This byte must contain a "0" digit in the high nibble and the sign in the low nibble.

The length of a trailing numeric string can be 0. In this case, no storage is occupied by the string. If a destination operand is a zero-length trailing numeric string, the sign of the operation is lost. Memory access faults will not occur when a zero-length trailing numeric operand is specified, because no memory reference occurs. The value of a zero-length trailing numeric string is identically 0.

The length of a leading separate numeric string can be 0. In this case, one byte of storage is occupied by the sign. Memory is accessed when a zero-length operand is specified, and a reserved operand abort occurs if an invalid sign is detected. The value of a zero-length leading separate numeric string is identically 0.

ADDP Add Packed

Format

```
opcode addlen.rw, addaddr.ab, sumlen.rw,        4 operand
       sumaddr.ab, {R0-3.wl}
opcode add1len.rw, add1addr.ab, add2len.rw,     6 operand
       add2addr.ab, sumlen.rw, sumaddr.ab,
       {R0-5.wl}
```

Operation

```
({sumaddr + ZEXT(sumlen/2)} : sumaddr) ←
   ({sumaddr + ZEXT(sumlen/2)} : sumaddr) +
   ({addaddr + ZEXT(addlen/2)} : addaddr);      !4 operand

({sumaddr + ZEXT(sumlen/2)} : sumaddr) ←
   ({add2addr + ZEXT(add2len/2)} : add2addr) +
   ({add1addr + ZEXT(add1len/2)} : add1addr);   !6 operand
```

Condition codes

```
N ← {sum string} LSS 0;
Z ← {sum string} EQL 0;
V ← {decimal overflow};
C ← 0;
```

Exceptions

```
reserved operand
decimal overflow
```

Opcodes

20	ADDP4	Add Packed 4 Operand
21	ADDP6	Add Packed 6 Operand

Description

In 4 operand format, the addend string specified by the addend length and addend address operands is added to the sum string specified by the sum length and sum address operands, and the sum string is replaced by the result.

In 6 operand format, the addend 1 string specified by the addend 1 length and addend 1 address operands is added to the addend 2 string specified by the addend 2 length and addend 2 address operands. The sum string specified by the sum length and sum address operands is replaced by the result.

1. After execution of ADDP4:
   - R0 = 0
   - R1 = address of the byte containing the most significant digit of the addend string
   - R2 = 0
   - R3 = address of the byte containing the most significant digit of the sum string

2. After execution of ADDP6:
   - R0 = 0
   - R1 = address of the byte containing the most significant digit of the addend 1 string
   - R2 = 0
   - R3 = address of the byte containing the most significant digit of the addend 2 string
   - R4 = 0
   - R5 = address of the byte containing the most significant digit of the sum string

3. The sum string, R0 through R3 (or R0 through R5 for ADDP6), and the condition codes are UNPREDICTABLE if the sum string overlaps the addend, addend 1, or addend 2 strings; the addend, addend 1, addend 2 or sum (4 operand only) strings contain an invalid nibble; or a reserved operand abort occurs.
4. The ADDPx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

ASHP Arithmetic Shift and Round Packed

Format

```
opcode cnt.rb, srclen.rw, srcaddr.ab, round.rb
       dstlen.rw, dstaddr.ab, {R0-3.wl}
```

Operation

```
({dstaddr + ZEXT(dstlen/2)} : dstaddr) ←
   {({srcaddr + ZEXT(srclen/2)} : srcaddr)
      + {round <3:0>*{10 ** {-cnt-1}}}}
      * {10 ** cnt} ;
```

Condition codes

```
N ← {dst string} LSS 0;
Z ← {dst string} EQL 0;
V ← {decimal overflow};
C ← 0;
```

Exceptions

```
reserved operand
decimal overflow
```

Opcodes

F8 ASHP Arithmetic Shift and Round Packed

Description

The source string specified by the source length and source address operands is scaled by a power of 10 specified by the count operand. The destination string specified by the destination length and destination address operands is replaced by the result.

A positive count operand effectively multiplies; a negative count effectively divides; and a zero count just moves and affects condition codes. When a negative count is specified, the result is rounded using the round operand.

Notes

1. After execution:
   - R0 = 0
   - R1 = address of the byte containing the most significant digit of the source string
   - R2 = 0
   - R3 = address of the byte containing the most significant digit of the destination string
2. The destination string, R0 through R3, and the condition codes are UNPREDICTABLE if the destination string overlaps the source string, the source string contains an invalid nibble, or a reserved operand abort occurs.
3. When the count operand is negative, the result is rounded by decimally adding bits ⟨3:0⟩ of the round operand to the most significant low-order digit discarded and propagating the carry, if any, to higher order digits. Both the source operand and the

round operand are considered to be quantities of the same sign for the purpose of this addition.

4. If bits ⟨7:4⟩ of the round operand are non-zero or if bits ⟨3:0⟩ of the round operand contain an invalid packed decimal digit, the result is UNPREDICTABLE.
5. When the count operand is zero or positive, the round operand has no effect on the result except as specified in note 4, above.
6. The round operand is normally five. Truncation may be accomplished by using a zero round operand.
7. The ASHP instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

CMPP Compare Packed

Format

```
opcode len.rw, src1addr.ab, src2addr.ab,        3 operand
       {R0-3.wl}
opcode src1len.rw, src1addr.ab, src2len.rw,     4 operand
       src2addr.ab, {R0-3.wl}
```

Operation

```
({src1addr + ZEXT(len/2)} : src1addr) -          !3 operand
   ({src2addr + ZEXT(len/2)} : src2addr);

({src1addr + ZEXT(src1len/2)} : src1addr) -      !4 operand
   ({src2addr + ZEXT(src2len/2)} : src2addr);
```

Condition codes

```
N ← {src1 string} LSS {src2 string};
Z ← {src1 string} EQL {src2 string};
V ← 0;
C ← 0;
```

Exceptions

reserved operand

Opcodes

35	CMPP3	Compare Packed 3 Operand
37	CMPP4	Compare Packed 4 Operand

Description

In 3 operand format, the source 1 string specified by the length and source 1 address operands is compared to the source 2 string specified by the length and source 2 address operands. The only action is to affect the condition codes.

In 4 operand format, the source 1 string specified by the source 1 length and source 1 address operands is compared to the source 2 string specified by the source 2 length and source 2 address operands. The only action is to affect the condition codes.

Notes

1. After execution of CMPP3 or CMPP4:
   - R0 = 0
   - R1 = address of the byte containing the most significant digit of string 1
   - R2 = 0
   - R3 = address of the byte containing the most significant digit of string 2
2. R0 through R3 and the condition codes are UNPREDICTABLE if the source strings overlap, if either string contains an invalid nibble, or if a reserved operand abort occurs.
3. The CMPPx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

CVTLP Convert Long to Packed

Format

```
opcode src.rl, dstlen.rw, dstaddr.ab, {R0-3.wl}
```

Operation

```
({dstaddr + ZEXT(dstlen/2)} : dstaddr) ← conversion of src;
```

Condition codes

```
N ← {dst string} LSS 0;
Z ← {dst string} EQL 0;
V ← {decimal overflow};
C ← 0;
```

Exceptions

```
reserved operand
decimal overflow
```

Opcodes

F9 CVTLP Convert Long to Packed

Description

The source operand is converted to a packed decimal string, and the destination string operand specified by the destination length and destination address operands is replaced by the result.

Notes

1. After execution:
   - R0 = 0
   - R1 = 0
   - R2 = 0
   - R3 = address of the byte containing the most significant digit of the destination string
2. The destination string, R0 through R3, and the condition codes are UNPREDICTABLE on a reserved operand abort.
3. Overlapping operands produce correct results.
4. The CVTLP instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

CVTPL Convert Packed to Long

Format

```
opcode srclen.rw, srcaddr.ab, {R0-3.wl}, dst.wl
```

Operation

```
dst ← conversion of ({srcaddr + ZEXT(srclen/2)} : srcaddr);
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {integer overflow};
C ← 0;
```

Exceptions

```
reserved operand
integer overflow
```

Opcodes

36 CVTPL Convert Packed to Long

Description

The source string specified by the source length and source address operands is converted to a longword, and the destination operand is replaced by the result.

Notes

1. After execution:
   - R0 = 0
   - R1 = address of the byte containing the most significant digit of the source string
   - R2 = 0
   - R3 = 0
2. The destination operand, R0 through R3, and the condition codes are UNPREDICTABLE on a reserved operand abort or if the string contains an invalid nibble.
3. The destination operand is stored after the registers are updated as specified in note 1, above. Thus, R0 through R3 may be used as the destination operand.
4. When the source string has a value that is outside the range –2,147,483,648 through +2,147,483,647, then integer overflow occurs and the destination operand is replaced by the low-order 32 bits of the correctly signed infinite precision conversion. Thus, on overflow, the sign of the destination may be different from the sign of the source.
5. Overlapping operands produce correct results.
6. The CVTPL instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

CVTPS Convert Packed to Leading Separate Numeric

Format

```
opcode srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab,
       {R0-3.wl}
```

Operation

```
{dst string} ← conversion of {src string};
```

Condition codes

```
N ← {src string} LSS 0;
Z ← {src string} EQL 0;
V ← {decimal overflow};
C ← 0;
```

Exceptions

```
reserved operand
decimal overflow
```

Opcodes

08 CVTPS Convert Packed to Leading Separate Numeric

Description

The source packed decimal string specified by the source length and source address operands is converted to a leading separate numeric string. The destination string specified by the destination length and destination address operands is replaced by the result.

Conversion is effected by replacing the lowest addressed byte of the destination string with the ASCII character "+" or "–", determined by the sign of the source string. The remaining bytes of the destination string are replaced by the ASCII representations of the values of the corresponding packed decimal digits of the source string.

Notes

1. After execution:
   - R0 = 0
   - R1 = address of the byte containing the most significant digit of the source string
   - R2 = 0
   - R3 = address of the sign byte of the destination string
2. The destination string, R0 through R3, and the condition codes are UNPREDICTABLE if the destination string overlaps the source string, the source string contains an invalid nibble, or a reserved operand abort occurs.
3. This instruction produces an ASCII "+" or "–" in the sign byte of the destination string.
4. If decimal overflow occurs, the value stored in the destination may be different from the value indicated by the condition codes (Z and PSL⟨N⟩ bits).

5. If the conversion produces a –0 without overflow, the destination leading separate numeric string is changed to a +0 representation.
6. The CVTPS instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

CVTPT Convert Packed to Trailing Numeric

Format

```
opcode srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw,
       dstaddr.ab, {R0-3.wl}
```

Operation

```
{dst string} ← conversion of {src string};
```

Condition codes

```
N ← {src string} LSS 0;
Z ← {src string} EQL 0;
V ← {decimal overflow};
C ← 0;
```

Exceptions

```
reserved operand
decimal overflow
```

Opcodes

24 CVTPT Convert Packed to Trailing Numeric

Description

The source packed decimal string specified by the source length and source address operands is converted to a trailing numeric string. The destination string specified by the destination length and destination address operands is replaced by the result. PSL⟨N⟩ and PSL⟨Z⟩ are affected by the value of the source packed decimal string.

Conversion is effected by using the highest addressed byte (even if the source string value is –0) of the source string (the byte containing the sign and the least significant digit) as an unsigned index into a 256-byte table whose zeroth entry address is specified by the table address operand. The byte read out of the table replaces the least significant byte of the destination string. The remaining bytes of the destination string are replaced by the ASCII representations of the values of the corresponding packed decimal digits of the source string.

Notes

1. After execution:
   - R0 = 0
   - R1 = address of the byte containing the most significant digit of the source string
   - R2 = 0
   - R3 = address of the most significant digit of the destination string

2. The destination string, R0 through R3, and the condition codes are UNPREDICTABLE if the destination string overlaps

the source string or the table, the source string or the table contains an invalid nibble, or a reserved operand abort occurs.

3. The condition codes are computed on the value of the source string even if overflow results. In particular, PSL⟨N⟩ is set if and only if the source is non-zero and contains a minus sign.
4. By appropriate specification of the table, conversion to any form of trailing numeric string may be realized. See Chapter 1 for the preferred form of trailing overpunched, zoned, and unsigned data. In addition, the table may be set up for absolute value, negative absolute value, or negated conversions. The translation table may be referenced even if the length of the destination string is zero.
5. Decimal overflow occurs if the destination string is too short to contain the converted result of a non-zero packed decimal source string (not including leading zeros). Conversion of a source string with zero value never results in overflow. Conversion of a non-zero source string to a zero-length destination string results in overflow.
6. If decimal overflow occurs, the value stored in the destination may be different from the value indicated by the condition codes (Z and PSL⟨N⟩ bits).
7. The CVTPT instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

CVTSP Convert Leading Separate Numeric to Packed

Format

```
opcode srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab,
       {R0-3.wl}
```

Operation

```
{dst string} ← conversion of {src string}
```

Condition codes

```
N ← {dst string} LSS 0;
Z ← {dst string} EQL 0;
V ← {decimal overflow};
C ← 0;
```

Exceptions

```
reserved operand
decimal overflow
```

Opcodes

09 CVTSP Convert Leading Separate Numeric to Packed

Description

The source numeric string specified by the source length and source address operands is converted to a packed decimal string, and the destination string specified by the destination address and destination length operands is replaced by the result.

Notes

1. A reserved operand abort occurs if:
   - The length of the source leading separate numeric string is outside the range 0 through 31
   - The length of the destination packed decimal string is outside the range 0 through 31
   - The source string contains an invalid byte; an invalid byte is any character other than an ASCII "0" through "9" in a digit byte or an ASCII "+", "⟨space⟩", or "–" in the sign byte

2. After execution:
   - R0 = 0
   - R1 = address of the sign byte of the source string
   - R2 = 0
   - R3 = address of the byte containing the most significant digit of the destination string

3. The destination string, R0 through R3, and the condition codes are UNPREDICTABLE if the destination string overlaps the source string or a reserved operand abort occurs.
4. The CVTSP instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

CVTTP Convert Trailing Numeric to Packed

Format

```
opcode srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw,
       dstaddr.ab, {R0-3.wl}
```

Operation

```
{dst string} ← conversion of {src string}
```

Condition codes

```
N ← {dst string}LSS 0;
Z ← {dst string} EQL 0;
V ← {decimal overflow};
C ← 0;
```

Exceptions

```
reserved operand
decimal overflow
```

Opcodes

26 CVTTP Convert Trailing Numeric to Packed

Description

The source trailing numeric string specified by the source length and source address operands is converted to a packed decimal string, and the destination packed decimal string specified by the destination address and destination length operands is replaced by the result.

Conversion is effected by using the highest addressed (trailing) byte of the source string as an unsigned index into a 256-byte table whose zeroth entry is specified by the table address operand. The byte read out of the table replaces the highest addressed byte of the destination string (the byte containing the sign and the least significant digit). The remaining packed digits of the destination string are replaced by the low-order 4 bits of the corresponding bytes in the source string.

Notes

1. A reserved operand abort occurs if:
   - The length of the source trailing numeric string is outside the range 0 through 31
   - The length of the destination packed decimal string is outside the range 0 through 31
   - The source string contains an invalid byte; an invalid byte is any value other than ASCII "0" through "9" in any high-order byte (any byte except the least significant byte)
   - The translation of the least significant digit produces an invalid packed decimal digit or sign nibble

2. After execution:
   - R0 = 0
   - R1 = address of the most significant digit of the source string
   - R2 = 0
   - R3 = address of the byte containing the most significant digit of the destination string

3. The destination string, R0 through R3, and the condition codes are UNPREDICTABLE if the destination string overlaps the source string or the table, or a reserved operand abort occurs.
4. If the convert instruction produces a –0 without overflow, the destination packed decimal string is changed to a +0 representation, PSL⟨N⟩ is cleared and PSL⟨Z⟩ is set.
5. If the length of the source string is 0, the destination packed decimal string is set identically equal to 0, and the translation table is not referenced.
6. By appropriate specification of the table, conversion from any form of trailing numeric string may be realized. See Chapter 1 for the preferred form of trailing overpunched, zoned, and unsigned data. In addition, the table may be set up for absolute value, negative absolute value, or negated conversions.
7. If the table translation produces a sign nibble containing any valid sign, the preferred sign representation is stored in the destination packed decimal string.
8. The CVTTP instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

DIVP Divide Packed

Format

```
opcode divrlen.rw, divraddr.ab, divdlen.rw,
       divdaddr.ab, quolen.rw, quoaddr.ab,
       {R0-5.wl, -16(SP):-1(SP).wb}
```

Operation

```
({quoaddr + ZEXT(quolen/2)} : quoaddr) ←
   ({divdaddr + ZEXT(divdlen/2)} : divdaddr) /
   ({divraddr + ZEXT(divrlen/2)} : divraddr);
```

Condition codes

```
N ← {quo string} LSS 0;
Z ← {quo string} EQL 0;
V ← {decimal overflow};
C ← 0;
```

Exceptions

```
reserved operand
decimal overflow
divide by zero
```

Opcodes

27 DIVP Divide Packed

Description

The dividend string specified by the dividend length and dividend address operands is divided by the divisor string specified by the divisor length and divisor address operands. The quotient string specified by the quotient length and quotient address operands is replaced by the result.

Notes

1. This instruction allocates a 16-byte workspace on the stack. After execution, SP is restored to its original contents and the contents of (SP)–16:(SP)–1 are UNPREDICTABLE.
2. The division is performed such that
   - The absolute value of the remainder (which is lost) is less than the absolute value of the divisor.
   - The product of the absolute value of the quotient times the absolute value of the divisor is less than or equal to the absolute value of the dividend.
   - The sign of the quotient is determined by the rules of algebra from the signs of the dividend and the divisor. If the value of the quotient is zero, the sign is always positive.
3. After execution:
   - R0 = 0
   - R1 = address of the byte containing the most significant digit of the divisor string
   - R2 = 0

- R3 = address of the byte containing the most significant digit of the dividend string
- R4 = 0
- R5 = address of the byte containing the most significant digit of the quotient string

4. The quotient string, R0 through R5, and the condition codes are UNPREDICTABLE if the quotient string overlaps the divisor or dividend strings, the divisor, dividend, or quotient strings overlap the 16 bytes of temporary storage on the stack, the divisor or dividend string contains an invalid nibble, the divisor is 0, or a reserved operand abort occurs.
5. The DIVP instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

MOVP Move Packed

Format

```
opcode len.rw, srcaddr.ab, dstaddr.ab, {R0-3.wl}
```

Operation

```
({dstaddr + ZEXT(len/2)} : dstaddr) ←
   ({srcaddr + ZEXT(len/2)} : srcaddr);
```

Condition codes

```
N ← {dst string} LSS 0;
Z ← {dst string} EQL 0;
V ← 0;
C ← C;
```

Exceptions

reserved operand

Opcodes

34 MOVP Move Packed

Description

The destination string specified by the length and destination address operands is replaced by the source string specified by the length and source address operands.

Notes

1. After execution:
   - R0 = 0
   - R1 = address of the byte containing the most significant digit of the source string
   - R2 = 0
   - R3 = address of the byte containing the most significant digit of the destination string
2. The destination string, R0 through R3, and the condition codes are UNPREDICTABLE if the destination string overlaps the source string, the source string contains an invalid nibble, or a reserved operand abort occurs.
3. If the source is –0, the result is +0, PSL⟨N⟩ is cleared and PSL⟨Z⟩ is set.
4. The MOVP instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

MULP Multiply Packed

Format

```
opcode mulrlen.rw, mulraddr.ab, muldlen.rw,
       muldaddr.ab, prodlen.rw, prodaddr.ab,
       {R0-5.wl}
```

Operation

```
({prodaddr + ZEXT(prodlen/2)} : prodaddr) ←
   ({muldaddr + ZEXT(muldlen/2)} : muldaddr) *
   ({mulraddr + ZEXT(mulrlen/2)} : mulraddr);
```

Condition codes

```
N ← {prod string} LSS 0;
Z ← {prod string} EQL 0;
V ← {decimal overflow};
C ← 0;
```

Exceptions

```
reserved operand
decimal overflow
```

Opcodes

25 MULP Multiply Packed

Description

The multiplicand string specified by the multiplicand length and multiplicand address operands is multiplied by the multiplier string specified by the multiplier length and multiplier address operands. The product string specified by the product length and product address operands is replaced by the result.

Notes

1. After execution:
   - R0 = 0
   - R1 = address of the byte containing the most significant digit of the multiplier string
   - R2 = 0
   - R3 = address of the byte containing the most significant digit of the multiplicand string
   - R4 = 0
   - R5 = address of the byte containing the most significant digit of the product string
2. The product string, R0 through R5, and the condition codes are UNPREDICTABLE if the product string overlaps the multiplier or multiplicand strings, the multiplier or multiplicand strings contain an invalid nibble, or a reserved operand abort occurs.
3. The MULP instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

SUBP Subtract Packed

Format

```
opcode sublen.rw, subaddr.ab, diflen.rw,        4 operand
       difaddr.ab, {R0-3.wl}
opcode sublen.rw, subaddr.ab, minlen.rw,        6 operand
       minaddr.ab, diflen.rw, difaddr.ab,
       {R0-5.wl}
```

Operation

```
({difaddr + ZEXT(diflen/2)} : difaddr) ←        !4 operand
   ({difaddr + ZEXT(diflen/2)} : difaddr) -
   ({subaddr + ZEXT(sublen/2)} : subaddr);

({difaddr + ZEXT(diflen/2)} : difaddr) ←        !6 operand
   ({minaddr + ZEXT(minlen/2)} : minaddr) -
   ({subaddr + ZEXT(sublen/2)} : subaddr);
```

Condition codes

```
N ← {dif string} LSS 0;
Z ← {dif string} EQL 0;
V ← {decimal overflow};
C ← 0;
```

Exceptions

```
reserved operand
decimal overflow
```

Opcodes

22	SUBP4	Subtract Packed 4 Operand
23	SUBP6	Subtract Packed 6 Operand

Description

In 4 operand format, the subtrahend string specified by subtrahend length and subtrahend address operands is subtracted from the difference string specified by the difference length and difference address operands, and the difference string is replaced by the result.

In 6 operand format, the subtrahend string specified by the subtrahend length and subtrahend address operands is subtracted from the minuend string specified by the minuend length and minuend address operands. The difference string specified by the difference length and difference address operands is replaced by the result.

Notes

1. After execution of SUBP4:
   - R0 = 0
   - R1 = address of the byte containing the most significant digit of the subtrahend string
   - R2 = 0
   - R3 = address of the byte containing the most significant digit of the difference string

2. After execution of SUBP6:
   - R0 = 0
   - R1 = address of the byte containing the most significant digit of the subtrahend string
   - R2 = 0
   - R3 = address of the byte containing the most significant digit of the minuend string
   - R4 = 0
   - R5 = address of the byte containing the most significant digit of the difference string

3. The difference string, R0 through R3 (or R0 through R5 for SUBP6), and the condition codes are UNPREDICTABLE if the difference string overlaps the subtrahend or minuend strings; if the subtrahend, minuend, or difference (4 operand only) strings contain an invalid nibble; or if a reserved operand abort occurs.
4. The SUBPx instructions belong to an instruction group that is optional to implement. For more detail, refer to Chapter 11.

3.14 Edit Instruction

The edit instruction is designed to implement the common editing functions for handling fixed-format output. The instruction converts an input packed decimal number to an output character string, generating characters for the output. This operation is exemplified by a MOVE to a numeric edited (PICTURE) item in COBOL or PL/I, but the instruction can be used for other applications as well. When converting digits, options include leading zero fill, leading zero protection, insertion of floating sign, insertion of floating currency symbol, insertion of special sign representations, and blanking an entire field when it is zero.

The operands to the EDITPC instruction are an input packed-decimal string descriptor, a pattern specification, and the starting address of the output string. The packed decimal descriptor is a standard VAX operand pair of the length of the decimal string in digits (up to 31) and the starting address of the string. The pattern specification is the starting address of a pattern operation editing sequence that is interpreted in much the same way as are the normal instructions. The output string is described by only its starting address, because the pattern defines the length unambiguously.

While the EDITPC instruction is operating, it manipulates two character registers and the four condition codes. One character register contains the fill character. This is normally an ASCII blank but would be changed to asterisk for check protection. The other character register contains the sign character. Initially, this register contains either an ASCII blank or a minus sign, depending on the sign of the input. The value of the register can be changed to allow other sign representations such as plus/minus or plus/blank and can be manipulated to output special notations such as CR or DB. The sign register can also be changed to the currency sign to implement a floating currency sign. After execution, the condition codes contain the sign of the input (N), the presence of a zero source (Z), an overflow condition (V), and the presence of significant digits (C). PSL⟨N⟩ is determined at the start of the instruction and is not changed thereafter (except to correct a –0 input). The other condition codes are computed and updated as the instruction proceeds. When the EDITPC instruc-

tion terminates, registers R0 through R5 contain the conventional values after a decimal instruction.

The edit instruction is optional in implementations of the VAX architecture. Execution of the instruction, when omitted, results in an emulated instruction exception. Omitted instructions are emulated by Digital-supported operating systems. Emulation software may allocate and use current-mode stack space when executed. For more detail, refer to Chapter 11.

EDITPC Edit Packed to Character String

Format

```
opcode srclen.rw, srcaddr.ab, pattern.ab, dstaddr.ab,
       {R0-5.wl}
```

Operation

```
if srclen GTRU 31 then {reserved operand abort};
PSW<V,C> ← 0;
PSW<Z> ← 1;
PSW<N> ← {src has minus sign};
R0 ← srclen;
tmp1 ← R0;
R1 ← srcaddr;
R2<15:8> ← {if PSW<N> EQL 0 then " " else "-"} ! sign of src
                 !R2<7:0> is used for the fill character
R3 ← pattern;
R5 ← dstaddr;
exit_flag ← false;

while NOT exit_flag do
   begin
   {fetch pattern byte};
   {if pattern 0:4 no operand};
   {if pattern 40:47 increment R3 and
               fetch one byte operand};
   {if pattern 80:AF except 80, 90, A0
               operand is rightmost nibble};
   {else {reserved operand fault}};
   {perform pattern operator};
   if NOT exit_flag then {increment R3};
   end;

if R0 NEQ 0 then {reserved operand abort};
R0 ← tmp1;                !length of source string
R1 ← R1 - {tmp1/2}        !point to start of source string
R2 ← 0;
R4 ← 0;
```

```
if PSW<Z> EQL 1 then PSW<N> ← 0;
```

Condition codes

```
N ← {src string} LSS 0;                    !N ← 0 if src is -0
Z ← {src string} EQL 0;
V ← {decimal overflow};                    !non-zero digits lost
C ← {significance};
```

Exceptions

```
reserved operand
decimal overflow
```

Opcodes

38 EDITPC Edit Packed to Character String

Description

The destination string specified by the pattern and destination address operands is replaced by the edited version of the source string specified by the source length and source address operands. The editing is performed according to the pattern string starting at the address pattern and extending until a pattern end (EO$END) pattern operator is encountered. The pattern string consists of 1-byte pattern operators. Some pattern operators take no operands. Some take a repeat count which is contained in the right-most nibble of the pattern operator itself. The rest take a 1-byte operand that immediately follows the pattern operator. This operand is either an unsigned integer length or a byte character. The individual pattern operators are described in Tables 3.3 and 3.4, and on the following pages.

Notes

1. A reserved operand abort occurs if srclen GTRU 31.
2. The destination string is UNPREDICTABLE if the source string contains an invalid nibble, if the EO$ADJUST_INPUT operand is outside the range 1 through 31, if the source and destination strings overlap, or if the pattern and destination strings overlap.
3. After execution, the registers are as shown in Figure 3.9. If the destination string is UNPREDICTABLE, R0 through R5 and the condition codes are UNPREDICTABLE.
4. If PSL⟨V⟩ is set at the end and PSL⟨DV⟩ is enabled, a numeric overflow trap occurs unless the conditions in note 9 are satisfied.
5. The destination length is specified exactly by the pattern operators in the pattern string. If the pattern is incorrectly formed or it is modified during the execution of the instruction, the length of the destination string is UNPREDICTABLE.
6. If the source is –0, the result may be –0 unless a fixup pattern operator is included (EO$BLANK_ZERO or EO$REPLACE_SIGN).

Table 3.3
EDIT Pattern Operators

Class	Name	Operand	Summary
insert	EO$INSERT	char[a]	insert char, fill if insignificant
	EO$STORE_SIGN	none	insert sign
	EO$FILL	repeat[b]	insert fill
move	EO$MOVE	repeat	move digits, filling insignificant
	EO$FLOAT	repeat	move digits, floating sign
	EO$END_FLOAT	none	end floating sign
fixup	EO$BLANK_ZERO	length[c]	fill backward when zero
	EO$REPLACE_SIGN	length	replace with fill if -0
load	EO$LOAD_FILL	char	load fill character
	EO$LOAD_SIGN	char	load sign character
	EO$LOAD_PLUS	char	load sign character if positive
	EO$LOAD_MINUS	char	load sign character if negative
control	EO$SET_SIGNIF	none	set significance flag
	EO$CLEAR_SIGNIF	none	clear significance flag
	EO$ADJUST_INPUT	length	adjust source length
	EO$END	none	end edit

[a]One character.
[b]Repeat count in the range 1 through 15.
[c]Length in the range 1 through 255.

7. The contents of the destination string and up to one page of memory preceding it are UNPREDICTABLE if the length covered by EO$BLANK_ZERO or EO$REPLACE_SIGN is 0 or outside the destination string.
8. If more input digits are requested by the pattern than are specified, a reserved operand abort is taken with R0 = -1 and R3 = location of pattern operator that requested the extra digit. The condition codes and other registers are UNPREDICTABLE.
9. If fewer input digits are requested by the pattern than are specified, then a reserved operand abort is taken with R3 = location of EO$END pattern operator. The condition codes and other registers are UNPREDICTABLE.
10. On an unimplemented or reserved pattern operator, a reserved operand fault is taken with R3 = location of the faulting pattern operator. This fault may be continued as long as the defined register state is manipulated according to the pattern operator description and the state specified as implementation dependent is preserved. FPD is set and the condition codes and registers are as follows:

Table 3.4
EDIT Pattern-Operator Encoding

Encoding	Operator	Operand
00	EO$END	none
01	EO$END_FLOAT	none
02	EO$CLEAR_SIGNIF	none
03	EO$SET_SIGNIF	none
04	EO$STORE_SIGN	none
05–1F	Reserved to DIGITAL	
20–3F	Reserved for all time	
40	EO$LOAD_FILL	char is in next byte
41	EO$LOAD_SIGN	char is in next byte
42	EO$LOAD_PLUS	char is in next byte
43	EO$LOAD_MINUS	char is in next byte
44	EO$INSERT	char is in next byte
45	EO$BLANK_ZERO	unsigned length is in next byte
46	EO$REPLACE_SIGN	unsigned length is in next byte
47	EO$ADJUST_INPUT	unsigned length is in next byte
48–5F	Reserved to DIGITAL	
60–7F	Reserved to DIGITAL's customers	
80	Reserved to DIGITAL	
81–8F	EO$FILL	repeat count is ⟨3:0⟩
90	Reserved to DIGITAL	
91–9F	EO$MOVE	repeat count is ⟨3:0⟩
A0	Reserved to DIGITAL	
A1–AF	EO$FLOAT	repeat count is ⟨3:0⟩
B0–FE	Reserved to DIGITAL	
FF	Reserved for all time	

N = {src has minus sign}
Z = all source digits 0 so far
V = non-zero digits lost
C = significance
R0⟨31:16⟩ = –{count of source zeros to supply}
R0⟨15:0⟩ = remaining srclen⟨15:0⟩
R1 = current source location
R2⟨31:16⟩ = implementation dependent
R2⟨15:8⟩ = current contents of sign character register
R2⟨7:0⟩ = current contents of fill character register
R3 = location of edit pattern operator causing exception

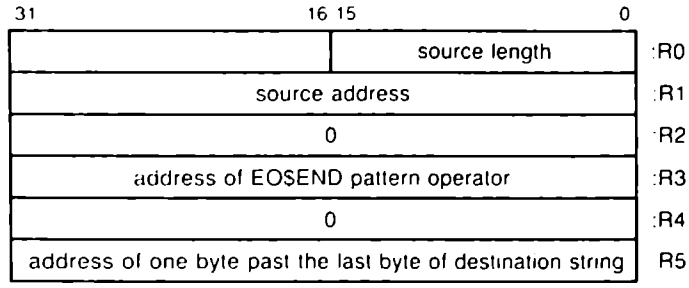


Figure 3.9
EDITPC Control Block

R4 = implementation dependent
R5 = location of next destination byte

11. The EDITPC instruction belongs to an instruction group that is optional to implement. For more detail, refer to Chapter 11.
12. The EDITPC instruction is in the emulated-only group and is unlikely to be implemented in future VAX processors. Software developers are advised to avoid using this instruction.

On the following pages, each pattern operator is defined in a format similar to that of the instruction descriptions. In each case, if there is an operand, it is a repeat count (r) from 1 through 15, an unsigned byte length (len), or a character byte (ch). In the formal descriptions, the following two routines are invoked:

```
READ: !function value 0 through 9
   if R0 EQL 0 then reserved operand;
   if R0 LSS 0 then
      begin
      READ ← 0;
      R0<31:16> ← R0<31:16> + 1;  !see EO$ADJUST_INPUT
      end;
   else
      begin
      READ ← (R1)<3+4*R0<0>:4*R0<0>>;  !get next nibble,
      R0 ← R0 - 1;                     !alternating high,
      if R0<0> EQL 1 then R1 ← R1 + 1; !then low.
      end;
   return;
STORE(char):
   (R5) ← char;
   R5 ← R5 + 1;
   return;
```

Further, the term "fill" refers to R2⟨7:0⟩, and "sign" to R2⟨15:8⟩.

EO$ADJUST_INPUT Adjust Input Length

Purpose

Handle source strings with lengths different from the output

Format

```
pattern len
```

Operation

```
if len EQLU 0 or len GTRU 31 then {UNPREDICTABLE};
if R0<15:0> GTRU len
then
   begin
   R0<31:16> ← 0
   repeat R0<15:0> - len do
      if READ NEQU 0 then
         begin
         PSW<Z> ← 0;
         PSW<C> ← 1;              !set significance
         PSW<V> ← 1;
         end;
   end;
else R0<31:16> ← R0<15:0> - len; !negative of number to fill
```

Pattern ops

47 EO$ADJUST_INPUT Adjust Input Length

Description

The pattern operator is followed by an unsigned byte integer length in the range 1 through 31. If the source string has more digits than this length, the excess leading digits are read and discarded. If any discarded digits are non-zero, then overflow is set, significance is set, and zero is cleared. If the source string has fewer digits than this length, a counter is set of the number of leading zeros to supply. This counter is stored as a negative number in R0(31:16).

Notes

1. If length is not in the range 1 through 31, the destination string, condition codes, and R0 through R5 are UNPREDICTABLE.

EO$BLANK_ZERO Blank Backwards When Zero

Purpose Fix the destination to be blank when the value is zero

Format

```
pattern len
```

Operation

```
if len EQLU 0 then {UNPREDICTABLE};
if PSW<Z> EQL 1 then
   begin
   R5 ← R5 - len;
   repeat len do STORE(fill);
   end;
```

Pattern ops 45 EOSBLANK_ZERO Blank Backwards When Zero

Description The pattern operator is followed by an unsigned byte integer length. If the value of the source string is zero, then the contents of the fill register are stored into the last length bytes of the destination string.

Notes

1. The length must be non-zero and within the destination string already produced. If it is not, the contents of the destination string and up to one page of memory preceding it are UNPREDICTABLE.
2. This pattern operator is used to blank out any characters stored in the destination under a forced significance, such as a sign or the digits following the radix point.

EO$END End Edit

Purpose End the edit operation

Format

```
pattern
```

Operation

```
exit_flag ← true;    !terminate edit loop
                     !end processing is
                     !described under EDITPC instruction
```

Pattern ops 00 EO$END End Edit

Description The edit operation is terminated.

Notes

1. If there are still input digits, a reserved operand abort is taken.
2. If the source value is –0, PSL⟨N⟩ is cleared.

EO$END_FLOAT End Floating Sign

Purpose End a floating sign operation

Format

```
pattern
```

Operation

```
if PSW<C> EQL 0 then
   begin
   STORE(sign);
   PSW<C> ← 1;          !set significance
   end;
```

Pattern ops 01 EO$END_FLOAT End Floating Sign

Description If the floating sign has not yet been placed in the destination (that is, if significance is not set), the contents of the sign register are stored in the destination and significance is set.

Notes

1. This pattern operator is used after a sequence of one or more EO$FLOAT operators that start with significance clear. The EO$FLOAT sequence can include intermixed EO$INSERT and EO$FILL operators.

EO$FILL Store Fill

Purpose Insert the fill character

Format

```
pattern r
```

Operation

```
repeat r do STORE(fill);
```

Pattern ops 8x EO$FILL Store Fill

Description The right nibble of the pattern operator is the repeat count. The contents of the fill register are placed into the destination repeat times.

Notes

1. This pattern operator is used for fill (blank) insertion.

EO$FLOAT Float Sign

Purpose

Move digits, floating the sign across insignificant digits

Format

```
pattern r
```

Operation

```
repeat r do
   begin
   tmp ← READ;
   if tmp NEQU 0 then
      if PSW<C> EQL 0 then
         begin
         STORE(sign);
         PSW<Z> ← 0;
         PSW<C> ← 1;          !set significance
         end;
   if PSW<C> EQL 0 then STORE(fill) else STORE("0" + tmp);
   end;
```

Pattern ops

Ax EO$FLOAT Float Sign

Description

The right nibble of the pattern operator is the repeat count. For repeat times, the following algorithm is executed. The next digit from the source is examined. If it is non-zero and significance is not yet set, then the contents of the sign register are stored in the destination, significance is set, and zero is cleared. If the digit is significant, it is stored in the destination; otherwise, the contents of the fill register are stored in the destination.

Notes

1. If r is greater than the number of digits remaining in the source string, a reserved operand abort is taken.
2. This pattern operator is used to move digits with a floating arithmetic sign. The sign must already be set up as for EO$STORE_SIGN. A sequence of one or more EO$FLOATs can include intermixed EO$INSERTs and EO$FILLs. Significance must be clear before the first pattern operator of the sequence. The sequence must be terminated by one EO$END_FLOAT.
3. This pattern operator is used to move digits with a floating currency sign. The sign must already be set up with an EO$LOAD_SIGN. A sequence of one or more EO$FLOATs can include intermixed EO$INSERTs and EO$FILLs. Significance must be clear before the first pattern operator of the sequence. The sequence must be terminated by one EO$END_FLOAT.

EO$INSERT Insert Character

Purpose

Insert a fixed character, substituting the fill character if significance is not set.

Format

```
pattern ch
```

Operation

```
if PSW<C> EQL 1 then STORE(ch) else STORE(fill);
```

Pattern ops

44 EO$INSERT Insert Character

Description

The pattern operator is followed by a character. If significance is set, then the character is placed into the destination. If significance is not set, then the contents of the fill register are placed into the destination.

Notes

1. This pattern operator is used for blankable inserts (comma, for example) and fixed inserts (slash, for example). Fixed inserts require that significance be set (by EO$SET_SIGNIF or EO$END_FLOAT).

EO$LOAD_ Load Register

Purpose

Change the contents of the fill or sign register

Format

```
pattern ch
```

Operation

```
case pattern of
   EO$LOAD_FILL:  fill ← ch;
   EO$LOAD_SIGN:  sign ← ch;
   EO$LOAD_PLUS:  if PSW<N> EQL 0 then sign ← ch;
   EO$LOAD_MINUS: if PSW<N> EQL 1 then sign ← ch;
end;
```

Pattern ops

40	EO$LOAD_FILL	Load Fill Register
41	EO$LOAD_SIGN	Load Sign Register
42	EO$LOAD_PLUS	Load Sign Register If Plus
43	EO$LOAD_MINUS	Load Sign Register If Minus

Description

The pattern operator is followed by an arbitrary character. For EO$LOAD_FILL, this character is placed into the fill register. For EO$LOAD_SIGN, this character is placed into the sign register. For EO$LOAD_PLUS, this character is placed into the sign register if the source string has a positive sign. For EO$LOAD_MINUS, this character is placed into the sign register if the source string has a negative sign.

Notes

1. EO$LOAD_FILL is used to set up check protection (asterisk instead of space).
2. EO$LOAD_SIGN is used to set up a floating currency sign.
3. EO$LOAD_PLUS is used to set up a non-blank plus sign.
4. EO$LOAD_MINUS is used to set up a non-minus minus sign (such as CR, DB, or the PL/I +).

EO$MOVE Move Digits

Purpose

Move digits, filling for insignificant digits (leading zeros)

Format

```
pattern r
```

Operation

```
repeat r do
   begin
   tmp ← READ;
   if tmp NEQU 0 then
      begin
      PSW<Z> ← 0;
      PSW<C> ← 1;          !set significance
      end;
   if PSW<C> EQL 0 then STORE(fill)
      else STORE("0" + tmp);
   end;
```

Pattern ops

9x EOSMOVE Move Digits

Description

The right nibble of the pattern operator is the repeat count. For repeat times, the following algorithm is executed. The next digit is moved from the source to the destination. If the digit is non-zero, significance is set and zero is cleared. If the digit is not significant (that is, if it is a leading zero), it is replaced by the contents of the fill register in the destination.

Notes

1. If r is greater than the number of digits remaining in the source string, a reserved operand abort is taken.
2. This pattern operator is used to move digits without a floating sign. If leading zero suppression is desired, significance must be clear. If leading zeros should be explicit, significance must be set. A string of EO$MOVEs intermixed with EO$INSERTs and EO$FILLs will handle suppression correctly.
3. If check protection (*) is desired, EO$LOAD_FILL must precede the EOSMOVE.

EO$REPLACE_SIGN Replace Sign When Zero

Purpose

Fix the destination sign when the value is zero

Format

```
pattern len
```

Operation

```
if len EQLU 0 then {UNPREDICTABLE};
if PSW<Z> EQL 1 then (R5 - len) ← fill;
```

Pattern ops

46 EO$REPLACE_SIGN Replace Sign When Zero

Description

The pattern operator is followed by an unsigned byte integer length. If the value of the source string is zero (that is, if PSL⟨Z⟩ is set), then the contents of the fill register are stored into the byte of the destination string that is length bytes before the current position.

Notes

1. The length must be non-zero and within the destination string already produced. If it is not, the contents of the destination string and up to one page of memory preceding it are UNPREDICTABLE.
2. This pattern operator can be used to correct a stored sign (EO$END_FLOAT or EO$STORE_SIGN) if a minus was stored and the source value turned out to be zero.

EO$_SIGNIF Significance

Purpose Control the significance (leading zero) indicator

Format `pattern`

Operation

```
PSW<C> ← 0;                    !EO$CLEAR_SIGNIF
PSW<C> ← 1;                    !EO$SET_SIGNIF
```

Pattern ops

02	EO$CLEAR_SIGNIF	Clear Significance
03	EO$SET_SIGNIF	Set Significance

Description The significance indicator is set or cleared. This controls the treatment of leading zeros (leading zeros are zero digits for which the significance indicator is clear).

Notes

1. EO$CLEAR_SIGNIF is used to initialize leading zero suppression (EO$MOVE) or floating sign (EO$FLOAT) following a fixed insert (EO$INSERT with significance set).
2. EO$SET_SIGNIF is used to avoid leading zero suppression (before EO$MOVE) or to force a fixed insert (before EO$INSERT).

EO$STORE_SIGN Store Sign

Purpose Insert the sign character

Format `pattern`

Operation `STORE(sign);`

Pattern ops

04	EO$STORE_SIGN	Store Sign

Description The contents of the sign register are placed into the destination.

Notes

1. This pattern operator is used for any non-floating arithmetic sign. It should be preceded by either the EO$LOAD_PLUS or EO$LOAD_MINUS operator if the default sign convention is not desired.

4

Memory Management

Memory management consists of the hardware and software that control the allocation and use of physical memory. The effect of memory management is exemplified in a multiprogramming system where several processes may reside in physical memory at the same time. To ensure that one process will not affect other processes or the operating system, VAX architecture uses memory protection and multiple address spaces.

Four hierarchical access modes provide the memory access control, which further improves software reliability. These access modes are, from most to least privileged, kernel, executive, supervisor, and user. For each of the four access modes, protection is specified at the individual page level, where a page may be inaccessible, read-only, or read/write. Any location accessible to one mode is also accessible to all more privileged modes. Furthermore, for each access mode, any location that can be written can also be read.

Memory management provides the CPU with mapping information. First, the CPU generates virtual addresses when an image is executed. Before these addresses can be used to access instructions and data, however, they must be translated into physical addresses. Memory management software maintains tables of mapping information (page tables) that keep track of where each 512-byte virtual page is located in physical memory. The CPU uses this mapping information when it translates virtual addresses to physical addresses.

Memory management, then, is the scheme that provides both the memory protection and memory-mapping mechanisms of VAX architecture. Memory management accomplishes the following:

- Provides a large address space for instructions and data
- Allows data structures up to one gigabyte
- Provides convenient and efficient sharing of instructions and data
- Contributes to software reliability

A virtual memory system provides a large address space, yet allows programs to run on hardware with small memory configurations. Programs execute in an environment termed a process. The virtual memory system for VAX provides each process with a 4-billion-byte address space.

The virtual address space is divided into two equal-size spaces: the system address space and the per-process address space. The system address space is the same for all processes. It contains the operating system, which is written as callable procedures. Thus, all system code can be available to all other system and user code with a simple CALL. Each process has its own separate process address space. However, several processes may have access to the same page, thus providing controlled sharing.

4.1 *Virtual Address Space*

A virtual address is a 32-bit unsigned integer specifying a byte location in the address space. The programmer sees a linear array of 4,294,967,296 bytes. The virtual address space is broken into 512-byte units termed pages. The page is the unit of relocation, sharing, and protection.

This virtual address space is too large to be contained in generally available main memory. Memory management provides the mechanism to map the active part of the virtual address space to the available physical address space. Memory management also provides page protection between processes. The operating system controls the virtual-to-physical address mapping tables and saves the inactive but used parts of the virtual address space on the external storage media.

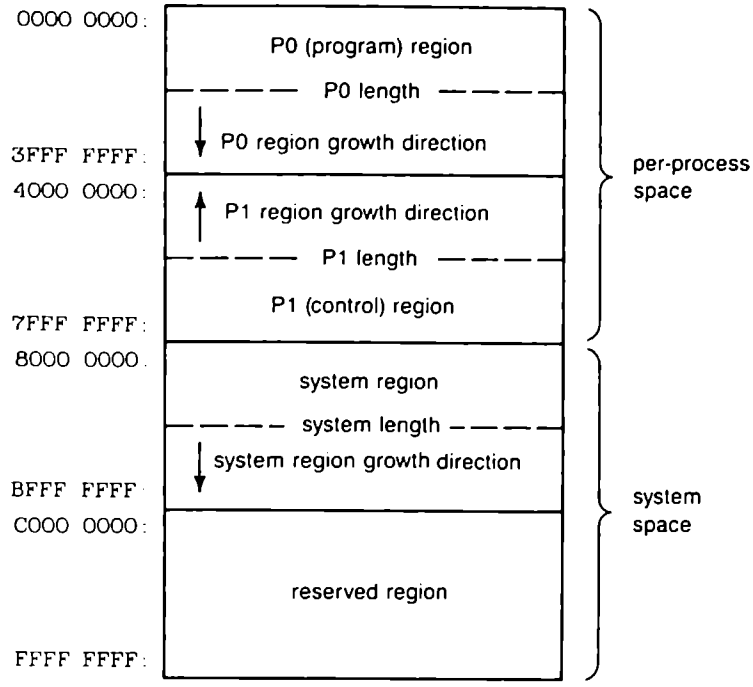


Figure 4.1
Virtual-Address Space

The virtual address space is divided into two parts, per-process space and system space, which are discussed in the following sections. Virtual address space is illustrated in Figure 4.1.

4.1.1 *Process Space*

The half of the virtual address space with smaller addresses (addresses 00000000 through 7FFFFFFF, hex) is termed per-process space. Per-process space is divided into two equal parts: the program region (P0 region) and the control region (P1 region). Each process has a separate address translation map for per-process space, so the per-process spaces of all processes are potentially completely disjoint. The address map for per-process space is context switched (changed) when the process running on the system is changed (see Chapter 6).

4.1.2 *System Space*

The half of virtual address space with larger addresses (addresses 80000000 through FFFFFFFF, hex) is termed system space. All processes use the same address translation map for system space,

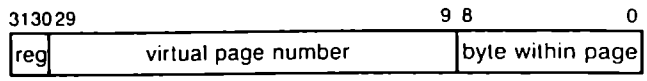


Figure 4.2
Virtual-Address Format

so system space is shared among all processes. The address map for system space is not context switched.

4.1.3 *Virtual Address Format*

The VAX processor generates a 32-bit virtual address for each instruction and operand in memory. As the process executes, the system translates each virtual address to a physical address. When memory management is enabled, the virtual address consists of a region field, a virtual page number (VPN) field, and a byte within page field, as shown in Figure 4.2 and described below.

The VPN field, bits ⟨29:9⟩ of a virtual address, specifies the virtual page to be referenced. The virtual address space contains 8,388,608 (2^{23}) pages. The byte-within-page field, bits ⟨9:0⟩ of a virtual address, specifies the byte offset within the page. A page contains 512 bytes.

The region field (bits ⟨31:30⟩ of a virtual address) is part of the virtual page number and specifies which of four regions the virtual address references. When bit ⟨31⟩ of a virtual address is 1, the address is in the system space. When bit ⟨31⟩ is 0, the address is in the per-process space.

Within system space, bit ⟨30⟩ distinguishes between the system region and a reserved region. When bits ⟨31:30⟩ are 11 (binary), the address refers to the reserved region. When bits ⟨31:30⟩ are 10 (binary), the address refers to the system region.

Within per-process space, bit ⟨30⟩ distinguishes between the program and control regions. When bits ⟨31:30⟩ are 01 (binary), the control region is referenced; and when bits ⟨31:30⟩ are 00, the program region is referenced.

4.1.4 *Virtual Address Space Layout*

The layout of virtual address space is illustrated in Figure 4.1. Note that access to each of the three regions (P0, P1, system)

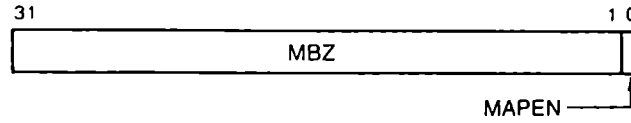


Figure 4.3
Map Enable Register (MAPEN)

is controlled by a length register (P0LR, P1LR, SLR). Within the limits set by the length registers, the access is further controlled by page tables that specify the validity, access requirements, and physical location of each page in the memory.

4.2 *Memory Management Control*

The action of translating a virtual address to a physical address is governed by the setting of the memory-mapping-enable (MME) bit in the MAPEN internal processor register. Figure 4.3 illustrates the privileged map-enable register.

MAPEN⟨0⟩ is the memory-mapping-enable bit. When MME is set to 1, memory management is enabled. When MME is set to 0, memory management is disabled. At processor initialization time, MAPEN is initialized to 0. Software should follow a MTPR-to-MAPEN with an REI instruction. (For further details, see section 7.2.2.)

4.2.1 *Memory Management Disabled*

When software disables memory mapping (by setting MME to 0), the processor turns off access control and translates virtual addresses to physical addresses by truncation, sign extension, or one-to-one mapping depending on the physical address mode implemented by the processor.

If the physical address mode that is implemented is 24-bit or 30-bit mode, virtual addresses are converted to 24-bit and 30-bit physical addresses, respectively, by truncating the high-order bits.

If the physical address mode is 32-bit one-to-one mode, the entire 32-bit virtual address is used as a 32-bit physical address. If the physical address mode is 32-bit sign-extended mode, a virtual address is converted by an implementation-dependent function,

Table 4.1
Physical Address Mode Selection

Feature	24-bit	30-bit	32-bit sign-extended	32-bit one-to-one	34-bit
Address size	PA⟨23:0⟩	PA⟨29:0⟩	PA⟨31:0⟩	PA⟨31:0⟩	PA⟨33:0⟩
Generated from:	VA⟨23:0⟩	VA⟨29:0⟩	SEXT(VA⟨29:0⟩)	VA⟨31:0⟩	SEXT(VA⟨29:0⟩)
PFN size	21-bit	21-bit	21-bit	25-bit	25-bit
Address in SBR	Phys	Phys	Phys	Phys	Phys
Address in SCBB	Phys	Phys	Phys	Phys	Phys/Virt
Address in PCBB	Phys	Phys/Virt	Phys	Phys	Virt
Range of SBR, SCBB, and PCBB used by CPU	⟨23:0⟩	⟨29:0⟩	⟨29:0⟩	⟨31:0⟩	⟨29:0⟩

which is preferably to sign extend VA bits ⟨29:0⟩ to 32 bits, and only part of the physical address space is visible when mapping is disabled. Whether a processor uses 32-bit one-to-one mode or 32-bit sign-extended mode depends on an implementation-dependent function, such as a mode-control bit.

If the physical address mode is 34 bit, virtual addresses are converted by an implementation-dependent function, and only part of the physical address space is visible when mapping is disabled. The preferred implementation of the 32- to 34-bit mapping is sign extension of VA bits ⟨29:0⟩.

The specific implementation of a number of memory management features is determined by the physical address mode that the processor supports. These features are described in subsequent sections and chapters. For convenience, Table 4.1 summarizes those implementation features that are allowed for each physical address mode.

When memory management is disabled, there is no page protection: all accesses are allowed in all modes. In addition, no modify bit is maintained, and no memory management faults occur.

The results of a reference to non-existent memory are implemen-

tation specific. References to addresses beyond the end of the physical address space result in UNDEFINED behavior. For example, a VAX-11/730, which implements a 24-bit physical address, may have 1 megabyte of memory installed. On such a system, a reference to an address in the first megabyte will read or write memory; a reference to an address in the second megabyte will result in a "non-existent memory" machine-check exception; and a reference to an address in the 17th megabyte will result in UNDEFINED processor operation.

Probing non-existent memory (or addresses beyond the end of the physical address space) results in UNDEFINED processor operation. In addition, a processor may have an instruction buffer that prefetches instructions before execution. If the instruction stream comes within 512 bytes of non-existent memory or the end of the physical address space when memory management is disabled, prefetcher references may cause UNDEFINED behavior. See Chapter 7 for additional information about physical memory and its operation.

4.3 *Address Translation*

When software enables memory mapping (by setting MME to 1), the processor turns access control on and translates virtual addresses to physical addresses by using page tables in memory. The processor uses the following to determine whether an intended access is allowed:

- The virtual address, which is used to index a page table
- The intended access type (read or write)
- The current privilege level from the processor status longword or the kernel level for page table mapping references

If the access is allowed and the address can be mapped (the page table entry is valid), the result is the physical address corresponding to the specified virtual address.

The intended access is READ if the operation to be performed is a read. The intended access is WRITE if the operation to be performed is a write. If the operation to be performed is a modify (that is, read followed by write), the intended access is specified as a WRITE.

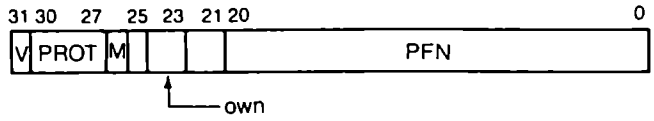


Figure 4.4
Page Table Entry (21-Bit Page Frame Number)

If an operand is an address operand, then no reference is made. Hence, the page need not be accessible and need not even exist.

4.3.1 *Page Table Entry*

The processor uses a page table entry (PTE) to translate virtual addresses to physical addresses. For a VAX processor with a physical address mode of either 24-bit, 30-bit, or 32-bit sign extended, the PTE format includes a 21-bit page-frame-number (PFN) field, as illustrated in Figure 4.4 and described in Table 4.2. For a VAX processor with a physical address mode of 32-bit one-to-one or 34-bit, the PTE format includes a 25-bit PFN field, as illustrated in Figure 4.5 and described in Table 4.3.

When PTE⟨V⟩ (the PTE valid bit, bit⟨31⟩) is zero, the PTE is said to be invalid. If software makes a reference to a page mapped by an invalid PTE (and the PTE protection field allows the access), the processor initiates a translation-not-valid fault. The operating system software uses some combinations of the software bits to implement its page-management data structures and functions. Among the functions implemented this way are initialize-pages-with-zeros, copy-on-reference, page sharing, and transitions between active and paged-out states. VAX/VMS encodes these functions in invalid PTEs and processes them whenever a page fault occurs.

The processor maintains (or helps software maintain) a record of pages that have been modified. When software attempts to write a page that is mapped by a valid PTE that allows the access, but PTE⟨M⟩ (the modify bit, bit⟨26⟩) is zero, the processor either initiates a modify fault and flushes the faulting PTE from the TB or sets PTE⟨M⟩ and continues, depending on the implementation. For process-space pages, only a zero value for process PTE⟨M⟩ causes the modify fault or the setting of process PTE⟨M⟩; the value of system PTE⟨M⟩ is ignored. Processors neither set PTE⟨M⟩ nor

Table 4.2
Fields of the PTE (21-Bit Page Frame Number)

Extent	Name	Mnemonic	Meaning
⟨31⟩	Valid	V	Indicates the validity of the M bit and PFN field. When V=1, the M and PFN fields are valid for use by hardware; when V=0, they are reserved for DIGITAL software.
⟨30:27⟩	Protection	PROT	Indicates at what access modes a process can reference the page. This field is always valid and is used by the CPU hardware even when V=0.
⟨26⟩	Modify	M	When V=0, M is not used by CPU hardware and is reserved for DIGITAL software and I/O devices. When V=1, M is used to detect write and modify references to the page.
⟨25⟩	Reserved	Z	Reserved to DIGITAL and must be zero.
⟨24:23⟩	Owner	OWN	Reserved for DIGITAL software.
⟨22:21⟩	Software	S	Reserved for DIGITAL software.
⟨20:0⟩	Page Frame Number	PFN	The upper 21 bits of the physical address of the base of the page. Used by CPU hardware only if V=1.

initiate modify fault if the page's protection denies access or if the PTE is invalid. Processors that implement modify fault never write the PTE.

Processors that do not implement modify fault set PTE⟨M⟩ as part of the result of a successful write or modify to a page (when mapping is enabled). Such processors may also set PTE⟨M⟩ as part of the result of a probe-write instruction (PROBEW) or as part of an implied probe-write.

When an instruction fault occurs on a processor that sets PTE⟨M⟩, it is UNPREDICTABLE whether the PTE⟨M⟩ bits are set for the valid and writable pages containing the instruction's write operands. For example, if a write reference crosses a page boundary where the first page is not accessible and the second page is accessible, the reference will fault. The processor does not set M in the PTE mapping the first page. It is UNPREDICTABLE whether the processor sets M in the PTE mapping the second page.

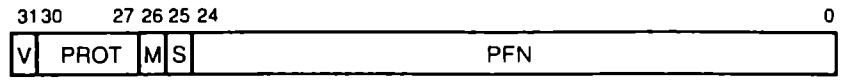


Figure 4.5
Page Table Entry (25-Bit Page Frame Number)

Table 4.3
Fields of the PTE (25-Bit Page Frame Number)

Extent	Name	Mnemonic	Meaning
⟨31⟩	Valid	V	Indicates the validity of the M bit and PFN field. When V=1, the M and PFN fields are valid for use by hardware; when V=0, they are reserved for DIGITAL software.
⟨30:27⟩	Protection	PROT	Indicates at what access modes a process can reference the page. This field is always valid and is used by the CPU hardware even when V=0.
⟨26⟩	Modify	M	When V=0, M is not used by CPU hardware and is reserved for DIGITAL software. When V=1, M is used to detect write and modify references to the page.
⟨25⟩	Software	S	Reserved for DIGITAL software.
⟨24:0⟩	Page Frame Number	PFN	The upper 25 bits of the physical address of the base of the page. Used by CPU hardware only if V=1.

It is UNPREDICTABLE whether the processor's setting a process PTE⟨M⟩ causes the processor to also set M in the system PTE that maps that process PTE.

Note that the update of PTE⟨M⟩ is not interlocked in a multiprocessor system.

4.3.2 *Changes to Page Table Entries*

The operating system changes PTEs as part of its memory management functions. For example, VAX/VMS sets and clears PTE⟨V⟩ and changes PTE⟨PFN⟩ as pages are moved to and from external storage devices.

Operating system software guarantees that each PTE is always consistent within itself. Changing a PTE one field at a time may give incorrect system operation. An example would be to set PTE⟨V⟩ with one instruction before establishing PTE⟨PFN⟩ with another. An interrupt routine between the two instructions could use an address that would map using the inconsistent PTE. The operating system can solve this problem by building a new PTE in a register and then moving the new PTE to the page table with a single instruction such as MOVL.

Multiprocessing makes the problem more complicated. Another processor, be it another CPU or an I/O processor, can reference the same page tables that the first CPU is changing. The second processor must always read consistent PTEs. To guarantee this, two requirements are met by VAX processors and operating system software (note that PTEs are longwords, longword-aligned):

1. Whenever the software modifies a PTE in more than one byte, it must use a longword, longword-aligned, and write-destination instruction such as MOVL.
2. A VAX processor must guarantee that a longword, longword-aligned write is an "atomic" operation. That is, a second processor cannot read (or write over) any of the first processor's partial results.

A processor writes a PTE only when setting that PTE's modify bit, and writes the PTE with a longword-aligned longword write.

4.4 Memory Protection

Memory protection is the function of validating whether a particular type of memory access is to be allowed to a particular page. Access to each page is controlled by a protection code that specifies for each access mode whether read or write references are allowed. Additionally, each address is checked to make certain that it lies within the P0, P1, or system region.

4.4.1 Processor Access Modes

In the order of most privileged to least privileged, the four processor modes are as follows:

0 Kernel mode: Page management, scheduling, I/O drivers

1 Executive mode: In VAX/VMS, many system-service calls, including RMS

2 Supervisor mode: In VAX/VMS, command interpretation

3 User mode: User-level code, utilities, compilers, debuggers

The access mode of a running process is the current processor mode, stored in the current-mode field of the processor status longword (PSL) (see section 1.3.2).

4.4.2 *Protection Code*

Every page in the virtual address space is protected according to its use. Even though all of the system space is shared, in the sense that all processes see the same system space, a program may be prevented from modifying or even reading portions of it. A program may also be prevented from reading or modifying portions of per-process space.

In system space, for example, scheduling queues are highly protected, whereas library routines may be executable by code of any privilege. Similarly, per-process accounting information may be in per-process space but highly protected, while normal user code in per-process space is executable at low privilege.

Associated with each page is a protection code that describes the accessibility of the page for each processor mode. The code allows a choice of protection for each processor mode, within the following limits:

- Each mode's access can be read-write, read-only, or no-access.
- If any level has read access, then all more privileged levels also have read access.
- If any level has write access, then all more privileged levels also have write access.
- If any level has write access, then it also has read access.

The protection codes for the 15 combinations of page protection are encoded in a 4-bit field in the page table entry, as shown in Table 4.4.

4.4.3 *Length Violation*

Every valid virtual address lies within bounds determined by the addressing region (P0, P1, or system) and the contents of the length register associated with that region (P0LR, P1LR, or SLR).

Table 4.4
PTE Protection Codes

Name	Mnemonic	Decimal	Binary	Accessibility[a]			
				Kernel	Exec	Super	User
no access	NA	0	0000	none	none	none	none
reserved		1	0001		UNPREDICTABLE		
kernel write	KW	2	0010	write	none	none	none
kernel read	KR	3	0011	read	none	none	none
user write	UW	4	0100	write	write	write	write
exec write	EW	5	0101	write	write	none	none
exec read / kernel write	ERKW	6	0110	write	read	none	none
exec read	ER	7	0111	read	read	none	none
super write	SW	8	1000	write	write	write	none
super read / exec write	SREW	9	1001	write	write	read	none
super read / kernel write	SRKW	10	1010	write	read	read	none
super read	SR	11	1011	read	read	read	none
user read / super write	URSW	12	1100	write	write	write	read
user read / exec write	UREW	13	1101	write	write	read	read
user read / kernel write	URKW	14	1110	write	read	read	read
user read	UR	15	1111	read	read	read	read

[a]The word "none" signifies no access is allowed; "read" signifies read access only; "write" signifies read and write access.

A reference to an address outside these bounds causes a length-violation fault. The addressing bounds algorithm is a simple limit check whose formal notation is:

```
case VAddr<31:30>
   [0]: if ZEXT(VAddr<29:9>) GEQU P0LR then ! P0 region
           {length violation};
   [1]: if ZEXT(VAddr<29:9>) LSSU P1LR then ! P1 region
           {length violation};
   [2]: if ZEXT(VAddr<29:9>) GEQU SLR then  ! System region
           {length violation};
   [3]: {length violation};                 ! reserved region
end;
```

4.4.4 *Access-Control-Violation Fault*

An access-control-violation fault occurs if an illegal access is attempted, as determined by the current PSL mode and the page's protection field, or if the address causes a length violation.

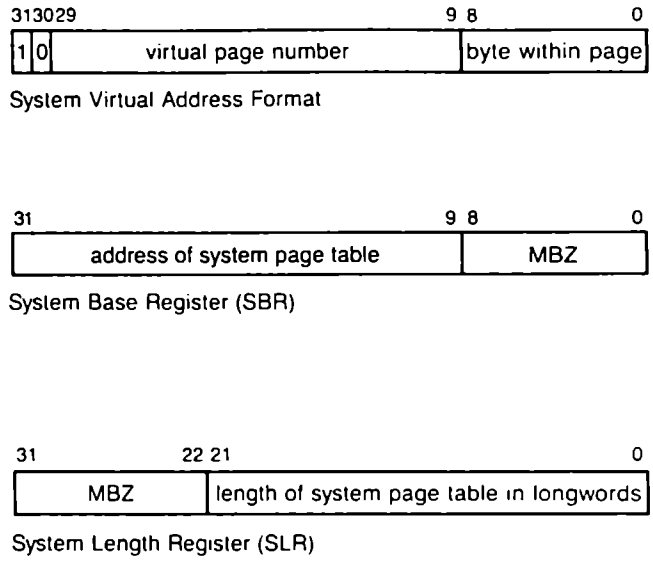


Figure 4.6
System Virtual-Address Space Registers

4.4.5 *Access Across a Page Boundary*

If an access is made across a page boundary, the order in which the pages are accessed is UNPREDICTABLE. For a single reference to a page, however, an access-control-violation fault always takes precedence over a translation-not-valid fault and modify fault.

4.5 *System-Space Address Translation*

A virtual address with ⟨31:30⟩ = 2 is an address in the system virtual address space. A system space address is shown in Figure 4.6.

The system virtual-address space is defined by the system page table, which is a physically contiguous table of page table entries. The system page table is a page-aligned data structure, located in either virtual or physical memory depending on the processor implementation. Its base address is contained in the system-base register (SBR), shown in Figure 4.6.

The SBR points to the first PTE in the system page table. In turn, this PTE maps the first page of system space, virtual addresses 80000000 through 800001FF (hex).

The size of the system page table in longwords (that is, the number of PTEs) is contained in the system-length register (SLR), shown in Figure 4.6. Bits ⟨31:9⟩ of the virtual address contain the virtual page number. However, system virtual addresses have VAddr⟨31:30⟩ = 2. Thus, there could be as many as 2^{21} pages in the system region. To express the values 0 through 2^{21} (200000 (hex)) inclusive, the length field in the SLR is 22 bits. SLR values not in the range of 0 to 200000 (hex), inclusive, are reserved values and result in UNDEFINED operation.

For all physical address modes, the processor locates the system page table in physical memory and holds a physical address in SBR. The processor uses this function to generate a physical address from a system region virtual address:

```
SYS_PA ← (SBR+4*SVA<29:9>)<PFN>'SVA<8:0>
```

Figures 4.7 and 4.8 illustrate the translation of a system virtual address to a physical address using this function.

To make it easier for processors to read aligned blocks of PTEs, software must pad the end of the system page table with invalid PTEs to a 32-byte boundary.

Processor initialization leaves the contents of SBR and SLR UNPREDICTABLE. If part or all of the system page table resides in I/O space, in non-existent memory, or beyond the end of the physical-address space while memory mapping is enabled, the operation of the processor is UNDEFINED.

The entire system page table must reside in the portion of physical memory that is accessible by virtual address translation when memory management is off. For example, on a processor that implements 34-bit physical address mode (as described in 4.2.1), software must ensure that the system page table resides within the first 512 megabytes of physical memory.

4.6 *Process-Space Address Translation*

The process virtual address space is divided into two separately mapped regions of equal size. If virtual address bit ⟨30⟩ is 0, the address is in region P0. If virtual address bit ⟨30⟩ is 1, the address is in region P1. Figure 4.9 illustrates a process virtual address.

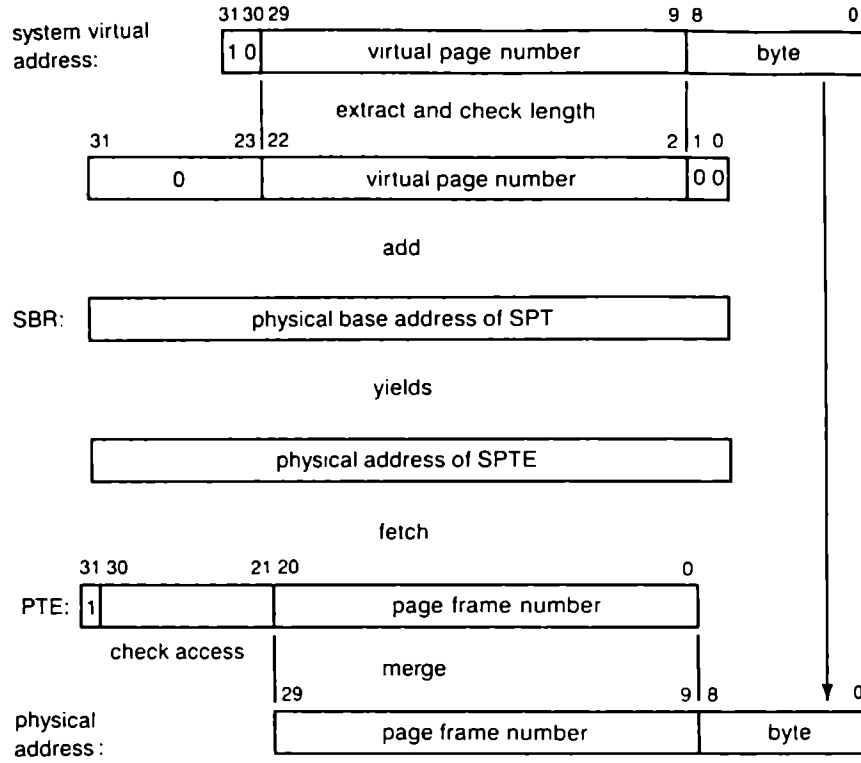


Figure 4.7
System Virtual Address to Physical Translation (21-Bit PFN)

The P0 region maps a virtually contiguous area that begins at the smallest address (0) in the process virtual-address space and grows in the direction of larger addresses. P0 is typically used for program images and can grow dynamically.

The P1 region maps a virtually contiguous area that begins at the largest address ($2^{31} - 1$) in the process virtual-address space and grows in the direction of smaller addresses. P1 is typically used for system-maintained, per-process context. It may grow dynamically for the user stack.

Each region is described by a contiguous vector of page table

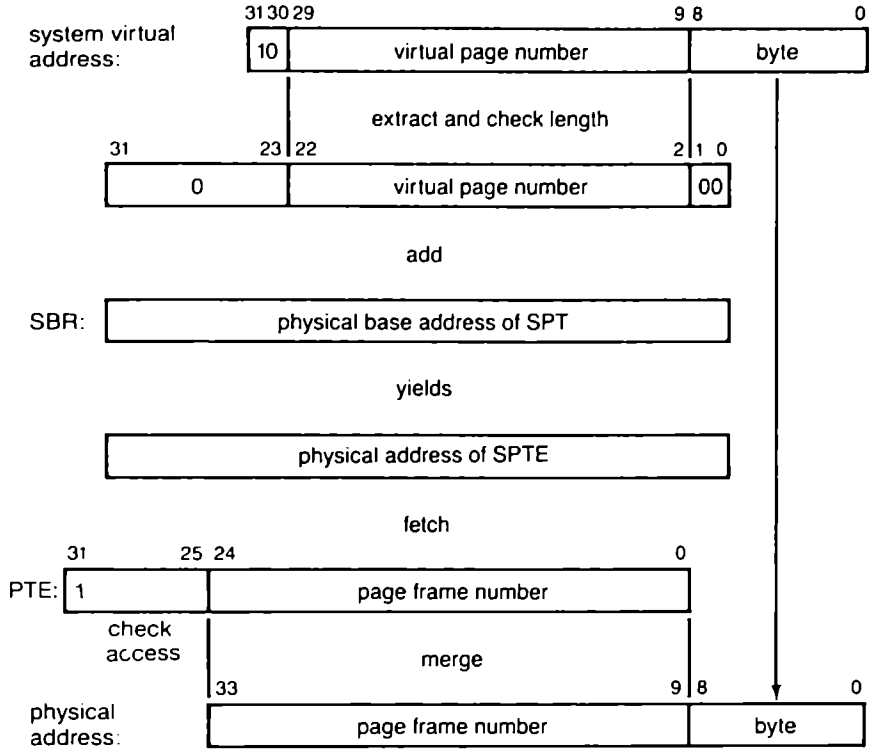


Figure 4.8
System Virtual Address to Physical Translation (25-Bit PFN)

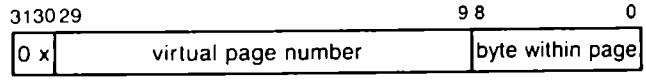


Figure 4.9
Process Space Virtual-Address Format

entries. Like the System Page Table, the process page tables may be addressed by either virtual or physical addresses, depending on the processor implementation. (A processor that implements process page tables in physical memory is known as an rtVAX variant. Such processors are described in Chapter 11.

There is reason to address process page tables in virtual rather than physical space: a physically addressed process page table that requires more than a page of PTEs (that is, that maps more than 64K bytes of process virtual-address space) requires physically

contiguous pages. Such a requirement makes dynamic allocation of process page table space awkward, since a running system tends to fragment storage into page-size areas.

There is also reason to address process page tables in physical memory: a process-space address translation that causes a translation-buffer miss will cause one memory reference for the process PTE. If the virtual address of the page containing the process PTE is also missing from the translation buffer, a second memory reference is required.

When a process page table entry is fetched by the processor from system space, the system space page containing the process PTE may be marked valid or invalid. If it is marked valid, the processor can read the process space PTE. If the system space page is invalid, a translation-not-valid fault results, and the "PTE reference" bit is set in the fault parameter. This allows the process page tables to be paged.

The operating system must make process page tables accessible to kernel mode, at least. The operation of the processor is UNDEFINED if process space page tables are read-only or no-access. Thus, the processor may or may not perform access checking (in kernel mode) when reading a process PTE or updating PTE⟨M⟩ in a process PTE.

When a process PTE is read from system space, a check is made against the system-page-table length register (SLR). Thus, the fetch of an entry from a process page table can result in either translation-not-valid or length-violation faults (see section 4.8).

If part or all of either process page table is mapped into I/O space or nonexistent memory while memory mapping is enabled, the operation of the processor is UNDEFINED.

4.6.1 *P0 Region*

The P0 region of the address space is mapped by the P0 page table (P0PT), which is defined by the P0 base register (P0BR) and the P0 length register (P0LR). The P0BR contains a page-aligned virtual address in the system region that is the base address of the P0PT. Figure 4.10 illustrates the P0BR. The P0LR contains

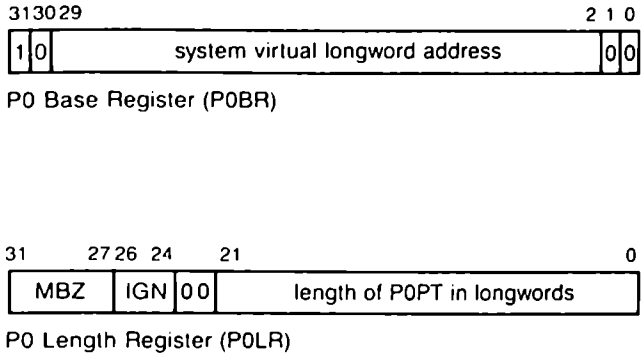


Figure 4.10
P0 Region Registers

the size of the P0PT in longwords, that is, the number of page table entries. Figure 4.10 also illustrates the P0LR. The page table entry addressed by the P0BR maps the first page of the P0 region of the virtual address space, that is, virtual byte address 0.

The PTEs in the P0PT contain the mapping information or may be used by system software to point to mapping information in a global page table. (See section 7.5.1 for this latter usage.)

The virtual page number is contained in bits ⟨29:9⟩ of the virtual address. There could be as many as 2^{21} pages in the P0 region. To express the values 0 through 2^{21} (200000 (hex)), inclusive, the length field in the P0LR is 22 bits. P0LR values not in the range of 0 to 200000 (hex), inclusive, are reserved values and result in UNDEFINED operation.

Because P0BR holds a page-aligned virtual address, P0BR⟨8:0⟩ must be zero (MBZ). In kernel mode, the operation of the processor is UNDEFINED after the execution of an MTPR to P0BR that writes a non-zero value to P0BR⟨8:0⟩.

An attempt to load P0BR with a value less than 2^{31} or greater than $2^{31} + 2^{30} - 4$ results in a reserved-operand fault in some implementations. Writing P0LR bits ⟨26:24⟩ has no effect. P0LR bits ⟨26:24⟩ read as zero. At processor initialization time, the contents of both P0BR and P0LR are UNPREDICTABLE.

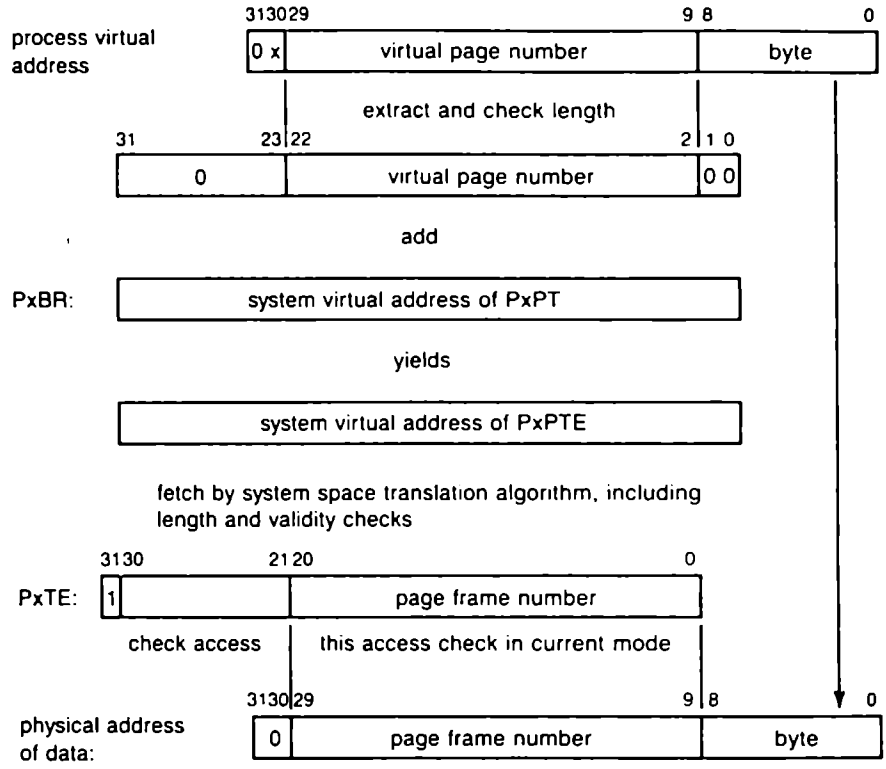


Figure 4.11
Process Virtual Address to Physical Translation (21-bit PFN)

The algorithm to generate a physical address from a P0 region virtual address is as follows:

```
P0PTE_VA ← P0BR+4*PVA<29:9>
P0PTE_PA ← (SBR+4*P0PTE_VA<29:9>)<PFN>'P0PTE_VA<8:0>
! Fetch from physical memory
P0_PA    ← (P0PTE_PA)<PFN>'P0_VA<8:0>
```

Figures 4.11 and 4.12 illustrate the translation of a process virtual address to a physical address using this function.

To make it easier for processors to read aligned blocks of PTEs, software must pad the end of the P0 page table with invalid PTEs to a 32-byte boundary.

4.6.2 *P1 Region*

The P1 region of the address space is mapped by the P1 page table (P1PT). P1PT is defined by the P1 base register (P1BR) and the P1

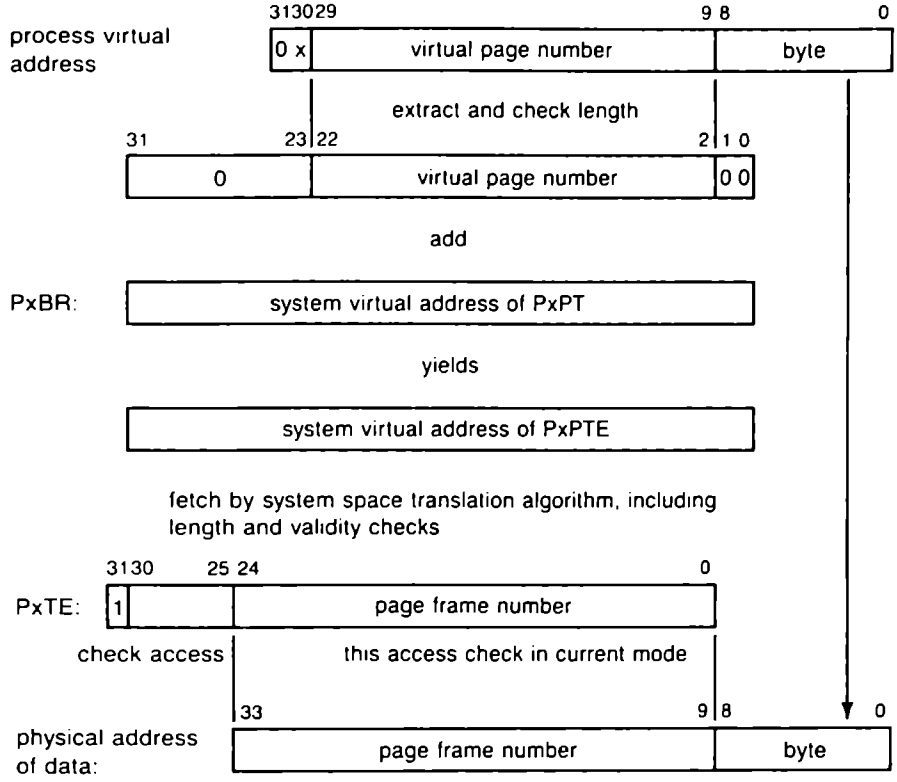


Figure 4.12
Process Virtual Address to Physical Translation (25-bit PFN)

length register (P1LR). Because P1 space grows toward smaller addresses, and because a consistent hardware interpretation of the base and length registers is desirable, P1BR and P1LR describe the portion of P1 space that is not accessible. Figure 4.13 illustrates the P1 base register and P1 length register. Note that P1LR contains the number of nonexistent PTEs. P1BR contains a page-aligned virtual address of what would be the PTE for the first page of P1, that is, virtual byte address 40000000 (hex).

The address in P1BR is not necessarily an address in system space, but all the addresses of PTEs must be in system space.

The PTEs in the P1PT contain the mapping information or may be used by system software to point to mapping information in a global page table. (See section 7.5.1 for this latter usage.)

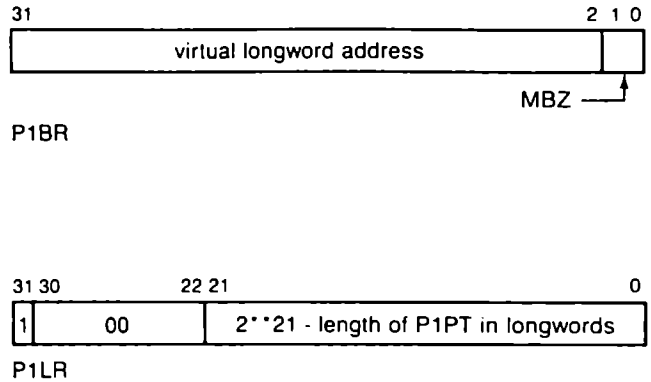


Figure 4.13
P1 Base and Length Registers

Because P1BR holds a page-aligned virtual address, P1BR⟨8:0⟩ must be zero (MBZ). In kernel mode, the operation of the processor is UNDEFINED after the execution of an MTPR to P1BR that writes a non-zero value to P1BR⟨8:0⟩. An attempt to load P1BR with a value less than $2^{31} - 2^{23}$ (7F800000, hex) or greater than $2^{31} + 2^{30} - 2^{23} - 4$ results in a reserved-operand fault in some implementations.

There could be from 0 to 2^{21} non-existent PTEs in the P1 page table. To express the values 0 through 2^{21} (200000 (hex)), inclusive, the length field in the P1LR is 22 bits. P1LR values not in the range of 0 to 200000 (hex), inclusive, are reserved values and result in UNDEFINED operation. Writing P1LR⟨31⟩ has no effect. The bit always reads as 0.

At processor initialization time, the contents of both P1BR and P1LR are UNPREDICTABLE.

The algorithm to generate a physical address from a P1 region virtual address is as follows:

```
P1PTE_VA ← P1BR+4*PVA<29:9>
P1PTE_PA ← (SBR+4*P1PTE_VA<29:9>)<PFN>'P1PTE_VA<8:0>
! Fetch from physical memory
P1_PA    ← (P1PTE_PA)<PFN>'P1_VA<8:0>
```

Figures 4.11 and 4.12 illustrate the process virtual address to physical address translation.

To make it easier for processors to read aligned blocks of PTEs, software must pad the beginning of the P1 page table with invalid PTEs from a 32-byte boundary. That is, the following number of longwords must immediately precede the longword at P1BR + 4*P1LR and contain PTEs with PTE⟨V⟩ clear: (P1LR modulo 8) longwords.

4.7 *Translation Buffer*

To avoid rereading PTEs when repeatedly translating references to the same virtual pages, a scalar or vector processor may include a mechanism to remember successful virtual address translations and page states. Such a mechanism is termed a translation buffer (TB). Any time a page table is active (that is, any time it is addressed by a page table base and length register, and MME=1), the processor may read and cache valid PTEs from that page table in a TB and may use them for translating addresses. The processor must not cache invalid PTEs. A cached PTE is also referred to as a TB entry.

Software is responsible for ensuring TB consistency by flushing stale PTEs from the TB. Between the times when software modifies a valid PTE in an active page table and when software flushes the old (stale) value of that PTE from any TBs that may have cached it, processors may use either the old value of the PTE or the new value of the PTE for translating references and for updating the PTE⟨M⟩ bit. Software must flush the entire TB before enabling memory mapping. Software can flush the entire TB on both the scalar and vector processors by writing zero to the translation-buffer-invalidate-all (TBIA) internal processor register. Software can flush the entire TB of just the vector processor by writing zero to the vector-translation-buffer-invalidate-all (VTBIA) internal processor register. Software can also flush the translation of a page mapped by an active page table from the TB of both the scalar and vector processors by writing any address in that page to the translation-buffer-invalidate-single (TBIS) internal processor register.

To save software the cost of explicit invalidates after software sets PTE⟨M⟩, processors that implement modify fault flush the faulting PTE from the TB as part of the process of initiating a modify fault. Note that the TB may reload and cache the old

PTE value between the time when modify fault flushes the old value and the time when software updates the PTE in memory. Software that depends on the hardware-provided flush must thus be prepared to take another modify fault on a page after setting the page's PTE⟨M⟩ bit. The second modify fault will flush the stale PTE from the TB, and the processor cannot load another stale copy. Thus, in the worst case, a multiprocessor system will take an initial modify fault and then an additional modify fault on each processor. In practice, even a single repetition is unlikely.

After software changes a valid system PTE that maps any part of an active process page table, software must flush the translation for the system page and then flush the translations of all valid process pages so mapped. If software writes such a process page before flushing its translation, the operation of the processor is UNDEFINED.

Because the TB must not store invalid PTEs, software is not required to flush TB entries after changing PTEs that were already invalid.

When software changes the location of a page table, it must flush translations as though it had modified all the PTEs the page table contained. When software reduces the size of a page table, it must flush translations as though it had modified all the PTEs that were part of the page table and are no longer.

Software is also responsible for ensuring that other processors in a multiprocessor do not use stale PTEs and that scalar/vector memory synchronization is performed before PTE modification. Software can use the following suggested protocol to properly coordinate invalidation:

1. Acquire exclusive access to the PTE: Prevent other processes, interrupt service routines, and software running on other processors from writing either the PTE or the page that the PTE maps. (Software must refrain from writing the page in order to prevent a processor from writing to the PTE to set PTE⟨M⟩.)
2. Ensure that scalar/vector memory synchronization is performed by each processor that has a vector processor present. (This ensures that vector processor loads and stores are no

longer executing and that all stores made by the vector processor are visible to its associated scalar processor.)

3. Update the PTE.
4. Flush the old value of the PTE from any TBs that may have cached it: If the PTE was invalid, then it couldn't have been cached in any TBs, so software needn't flush it. If a processor's TB has already been flushed since the last time the page table was active on that processor, there's no need to flush it again. But if the PTE was valid and part of a page table that was active on a particular processor, software must assume that the PTE may be cached in that processor's TB—even if software on that processor has made no explicit reference to the page—so software must flush the PTE from that processor's TB.
5. Release exclusive access to the PTE.

Other, more efficient, protocols are possible in restricted cases.

When software writes a virtual address to the Translation-Buffer Check (TBCHK) internal processor register, the processor may set the condition code V-bit if the TB contains a valid translation of that address, and clears the V-bit if the TB does not. (PSL⟨N⟩, PSL⟨Z⟩, and PSL⟨C⟩ are UNPREDICTABLE.) The results are useful for heuristic paging algorithms. Because the translation buffer can flush translations at any time, and can load PTEs from active page tables at any time, the results may be out of date by the time they are read, and so may not be useful for other purposes.

The specification of the TBCHK register is based on VAX/VMS usage. Its specification is subject to change without prior notice.

The TBIS, TBIA, and TBCHK processor registers are write only. The operation of MFPR from any of these registers is UNDEFINED.

Before enabling memory management, software must flush the entire translation buffer. If software does not follow this rule, the operation of the processor becomes UNDEFINED when memory management is enabled.

Note that vector processor memory access instructions must not be used to read or write active page tables.

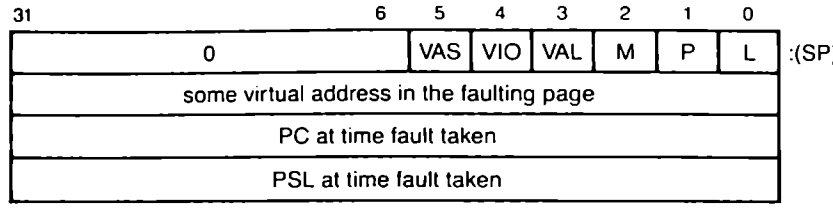


Figure 4.14
Memory-Management-Fault Stack Frame

4.8 *Faults and Parameters*

Three types of faults are associated with memory mapping and protection. A translation-not-valid fault is taken when a read or write reference is attempted through an invalid PTE (PTE⟨31⟩=0). An access-control-violation fault is taken when the protection field of the PTE indicates that the intended page reference in the specified access mode would be illegal. A modify fault is taken when a write or modify reference is attempted to a page that has not previously been modified (PTE⟨26⟩=0). Note that these three faults have distinct vectors in the system control block. If more than one type of fault could occur on a reference to a single page, then access-control-violation fault takes precedence over both translation-not-valid fault and modify fault. An access-control-violation fault is also taken if the virtual address referenced is beyond the end of the associated page table. Such a "length violation" is essentially the same as referencing a PTE that specifies "no access" in its protection field. The fault software does not have to compute the length check because a "length violation" indication is stored in the memory management fault stack frame, illustrated in Figure 4.14. See Chapter 5 for a description of faults.

The same parameters are stored for all the memory-management faults, including those of the vector processor. The first parameter pushed on the stack after the PSL and PC is some virtual address in the same page with the virtual address that caused the fault. A process-space reference can result in a system-space virtual reference for the PTE (except on an rtVAX processor). If the PTE reference faults, the virtual address that is saved is the process virtual address. In addition, a 1 is stored in bit ⟨1⟩ of the fault

Table 4.5
Fields of the Memory-Management-Fault Parameter

Name	Extent	Meaning
MBZ	⟨31:6⟩	Reserved to DIGITAL.
vector asynchronous MME	⟨5⟩	(VAS) Set to 1 to indicate that a vector processor memory management exception has occurred when the asynchronous memory management scheme is implemented.
vector I/O reference	⟨4⟩	(VIO) Set to 1 on some vector implementations to indicate that an access-control violation has occurred due to a vector instruction reference to I/O space.
vector alignment	⟨3⟩	(VAL) Set to 1 when a vector access-control violation has occurred due to a vector element not being properly aligned in memory.
modify or write intent	⟨2⟩	Set to 1 to indicate that the instruction's intended access was write or modify. This bit is 0 if the instruction's intended access was read. This bit is always 1 for a modify fault.
PTE reference	⟨1⟩	Set to 1 to indicate that the fault occurred during the reference to the process page table associated with the virtual address. This can be set on either translation-not-valid or length-violation faults. This bit is always 0 for a modify fault, and on an rtVAX.
length violation	⟨0⟩	Set to 1 to indicate that an access-control-violation fault was the result of a length violation rather than a protection violation. This bit is always 0 for a translation-not-valid fault and for a modify fault.

parameter word if the fault occurred in the per-process PTE reference. The fields of the second parameter are described in Table 4.5.

4.9 Privileged Services and Argument Validation

This section lists the instructions allowing access mode change and describes two instructions that allow privileged services to check addresses passed as parameters.

4.9.1 Changing Access Modes

Four instructions allow a program to change its access mode to a more privileged mode and transfer control to a service dispatcher for the new mode.

CHMK change mode to kernel
CHME change mode to executive
CHMS change mode to supervisor
CHMU change mode to user

These instructions provide the normal mechanism for less privileged code to call more privileged code; the instructions are described in detail in Chapter 5. When the mode transition takes place, the previous mode is saved in the previous-mode field of the PSL, thus allowing the more privileged code to determine the privilege of its caller.

4.9.2 *Validating Address Arguments*

Two instructions, PROBER and PROBEW, allow privileged services to check addresses passed as parameters. To avoid protection holes in the system, a service routine must always verify that its less privileged caller could have directly referenced the addresses passed as parameters. The PROBE instructions do this verification.

PROBEx Probe Accessibility

Format

```
opcode mode.rb, len.rw, base.ab
```

Operation

```
probe_mode ← MAXU (mode<1:0>, PSL<PRV_MOD>);
flag ← {accessibility of base} AND
       {accessibility of {base+ZEXT(len)-1}}
       using probe_mode;
```

Condition codes

```
N ← 0;
Z ← if {flag EQLU accessible} then 0 else 1;
V ← 0;
C ← C;
```

Exceptions

```
translation not valid
modify
```

Opcodes

0C	PROBER	Probe Read Accessibility
0D	PROBEW	Probe Write Accessibility

Description

The PROBE instruction checks the read or write accessibility of the first and last byte specified by the base address and the zero-extended length. Note that the bytes in between are not checked. System software must check all pages between the two end bytes if they will be accessed.

The protection is checked against the larger (and therefore less privileged) of the modes specified in bits ⟨1:0⟩ of the mode operand and the previous-mode field of the PSL. Note that probing with a mode operand of 0 is equivalent to probing the mode specified in PSL⟨PRV_MOD⟩.

1. If the valid bit of the examined PTE is set and write access is allowed, it is UNPREDICTABLE whether the modify bit of the examined PTE is set by a PROBEW. If the valid bit is clear or if write access is not allowed, the modify bit is not changed.
2. Except as in note 1, above, the processor ignores the valid bit of the PTE mapping the probed address.
3. A length violation gives a status of "not accessible."
4. During the probe of a process virtual address, on a non-rtVAX processor, if the valid bit of the system PTE is 0 then a translation-not-valid fault occurs. This allows for the demand paging of the process page tables. On an rtVAX, the process-space page tables are located in physical memory, and this fault cannot occur. (See section 11.2.3 for more information on rtVAX memory management.)
5. An object 1 byte long is the smallest that can be probed. With a length of zero, the PROBE instructions test the accessibility of two bytes (base and base − 1).
6. If memory management is disabled, all memory is accessible, and probing nonexistent memory gives UNPREDICTABLE results.

Example

```
MOVL    4(AP),R0        ; Copy the address of first
                        ; argument so that it can't be
                        ; changed.
PROBER  #0,#4,(R0)      ; Verify that the longword
                        ; pointed to by the first arg
                        ; could be read by the previous
                        ; access mode.  (Note that the
                        ; arg list itself must already
                        ; have been probed.)
BEQL    violation       ; Branch if either byte gives an
                        ; access violation.
MOVQ    8(AP),R0        ; Copy length and address of
                        ; buffer args so that they can't
                        ; change.
PROBEW  #0,R0,(R1)      ; Verify that the buffer
```

```
                        ; described by the second and
                        ; third args could be written by
                        ; the previous access mode.
                        ; (Note that the arg list must
                        ; already have been probed and
                        ; that the second arg must be
                        ; less than 512.)
BEQL    violation       ; Branch if either byte gives an
                        ; access violation.
```

Flows

The following description is of the operational flow of PROBE on each of the virtual addresses it is checking. Note that probing an address returns only the accessibility of the page(s) and has no effect on its residency. However, probing a process address may cause a page fault in the system address space on the per-process page tables.

1. Look up the virtual address in the TB. If found, use the associated protection field to determine the accessibility and EXIT.
2. Check for length violation for system or per-process address, as appropriate. See elsewhere in this chapter for the length-violation check flows. If length violation then return No Access and EXIT.
3. If per-process virtual address, go to step 4. Form physical address of PTE, fetch the PTE, and use the protection field to determine the accessibility. Then EXIT.
4. If per-process virtual address on an rtVAX, form physical address of PTE, fetch the PTE, use the protection field to determine the accessibility, and EXIT.
5. If process-space virtual address on a non-rtVAX processor, must do a virtual memory reference for the PTE.

   a. Look up the virtual address of the PTE in the TB. If not found, go to step b, below. Form the physical address of the PTE, fetch the PTE, and use the protection field to determine the accessibility. Then EXIT.
   b. If the virtual address of the PTE is not in the TB, check the system virtual address of the PTE for length violation. If length violation, then return No Access and EXIT.
   c. Read the SPTE for the system-space page containing the per-process PTE.

d. If the valid bit in the SPTE is 0, then take a translation-not-valid fault and EXIT. This case allows for the demand paging of per-process page tables.
e. Finally, calculate the physical address of the per-process PTE from the PFN field of the SPTE (see section 4.5), fetch the per-process PTE, and use the protection field to determine the accessibility. Then EXIT.

5

Interrupts and Exceptions

At certain times during the operation of a system, events within the system require the execution of particular pieces of software outside the explicit flow of control. The processor transfers control by forcing a change in the flow of control from that explicitly indicated in the currently executing process.

Some of the events are relevant primarily to the currently executing process and normally invoke software in the context of the current process. The notification of such an event is termed an "exception." Both the scalar and vector processors generate exceptions; however, they differ in the mechanism they use to notify the currently executing process. This chapter discusses exceptions generated by the scalar processor. Section 13.2 discusses exceptions generated by the vector processor.

Other events are primarily relevant to other processes or to the system as a whole and are therefore serviced in a system-wide context. The notification process for these events is termed an "interrupt," and the system-wide context is described as "executing on the interrupt stack." Further, some interrupts are of such urgency that they require high-priority service, while others must be synchronized with independent events. To meet these needs, the processor has priority logic that grants interrupt service to the highest priority event at any point in time. The priority associated with an interrupt is termed its interrupt priority level (IPL).

Processor Interrupt Priority Level and Interrupts

The processor has 31 interrupt priority levels: 15 software levels (numbered, in hex, 01 to 0F) and 16 hardware levels (10 to 1F, hex). User applications, system calls, and system services all run at process level, which may be thought of as IPL 0. Higher numbered interrupt levels have higher priority; that is to say, any requests at an interrupt level higher than the processor's current IPL will interrupt immediately, but requests at a lower or equal level are deferred.

Interrupt levels 01 through 0F (hex) exist entirely for use by software. No device can request interrupts on those levels, but software can force an interrupt by executing *MTPR src, #PR$_SIRR*. (See Chapter 8 and section 5.6.3.) Once a software interrupt request is made, it will be cleared by the hardware when the interrupt is taken.

Interrupt levels 10 to 17 (hex) are for use by devices and controllers, including UNIBUS devices if the system has a UNIBUS interconnect; UNIBUS levels BR4 to BR7 correspond to VAX interrupt levels 14 to 17 (hex). Interrupt levels 18 to 1F (hex) are for use by urgent conditions, serious errors, and powerfail.

The processor arbitrates interrupt requests according to priority. Only when the priority of an interrupt request is higher than the processor's current IPL (stored in PSL⟨20:16⟩) will the processor raise its IPL and service the interrupt request. The interrupt service routine is entered at the IPL of the interrupt request and usually will not change the IPL set by the processor. Note that this is different from the PDP-11, where the interrupt vector specifies the IPL for the interrupt service routine.

Interrupt requests can come from devices, controllers, other processors, or the processor itself. Software executing in kernel mode can raise and lower the priority of the processor by executing *MTPR src, #PR$_IPL*, where src contains the new priority desired. However, a processor cannot disable interrupts on other processors. Furthermore, the priority level of one processor does not affect the priority level of the other processors. Thus, in multiprocessor systems, interrupt priority levels cannot be used to

synchronize access to shared resources. Even the various urgent interrupts, including those exceptions that run at IPL 1F (hex), do so on only one processor. Consequently, special software action is required to stop other processors in a multiprocessor system.

Underlying the VAX architectural concept of an interrupt is the notion that an interrupt request is a static condition, not a transient event, which can be sampled by a processor at appropriate times. Further, if the need for an interrupt disappears before a processor has honored an interrupt request, the interrupt request can be removed (subject to implementation-dependent timing constraints) without consequence.

In order for software to operate deterministically, any instruction changing the processor priority (IPL) such that a pending interrupt is enabled must allow the interrupt to occur before executing the next instruction that would have been executed had the interrupt not been pending.

Similarly, instructions that generate requests at software interrupt levels must allow the interrupt to occur, if processor priority permits, before executing the apparently subsequent instruction.

5.2 *Exception Types*

Most exception service routines execute at IPL 0 in response to exception conditions caused by the software. A variation from this is serious system failures, which raise IPL to the highest level (1F, hex) to minimize processor interruption until the problem is corrected. Exception service routines are usually coded to avoid exceptions; however, nested exceptions can occur.

A trap is an exception that occurs at the end of the instruction that caused the exception. Therefore, the PC saved on the stack is the address of the next instruction that normally would have been executed. Any software can enable and disable some trap conditions by using the BISPSW and BICPSW instructions described in Chapter 3.

A fault is an exception that occurs during an instruction and that leaves the registers and memory in a consistent state such that

elimination of the fault condition and restarting the instruction will give correct results. After an instruction faults, the PC saved on the stack points to the instruction that faulted. Note that faults do not always leave everything as it was prior to the faulted instruction; they only restore enough to allow restarting. Thus, the state of a process that faults may not be the same as that of a process that was interrupted at the same point.

An abort is an exception that occurs during an instruction. An abort leaves the value of registers and memory UNPREDICTABLE such that the instruction cannot necessarily be correctly restarted, completed, simulated, or undone. After an instruction aborts, the PC saved on the stack points to the opcode of the aborted instruction. The following are UNPREDICTABLE:

- Destination operands (including implied operands, such as the top of the stack in a JSB instruction)
- Registers modified by operand specifier evaluation (including specifiers for implied operands)
- PTE⟨M⟩ in a PTE that maps a write-access or modify-access operand, where the implementation sets PTE⟨M⟩ rather than taking modify faults, and write access to the operand was allowed, and the operand was not written, and PTE⟨M⟩ was clear at the beginning of the instruction
- Condition codes
- PSL⟨FPD⟩
- PSL⟨TP⟩, if PSL⟨T⟩ was set at the beginning of the instruction

Except where otherwise noted in the description of the abort, the rest of the PSL, other registers, and memory are unaffected. Instructions that abort are constrained only by memory protection.

5.3 *Interrupts versus Exceptions*

Generally, exceptions and interrupts are very similar. When either is initiated, both the processor status longword and the program counter are pushed onto the stack. There are, however, several important differences:

- An exception condition is caused by the execution of the current instruction, whereas an interrupt is caused by some

activity in the computing system that may be independent of the current instruction.

- An exception condition usually is serviced in the context of the process that produced the exception condition, whereas an interrupt is serviced independently from the currently running process.
- The IPL of the processor usually is not changed when the processor initiates an exception, whereas the IPL is always raised when an interrupt is initiated.
- Exception service routines usually execute on a per-process stack, whereas interrupt service routines normally execute on a per-CPU stack.
- Enabled exceptions are always initiated immediately, no matter what the processor IPL is, whereas interrupts are held off until the processor IPL drops below the IPL of the requesting interrupt.
- Most exceptions cannot be disabled. However, if an exception-causing event occurs while that exception is disabled, no exception is initiated for that event even when enabled subsequently. This includes overflow, the only exception condition whose occurrence is indicated by a condition code (V). If an interrupt condition occurs while it is disabled, or the processor is at the same or higher IPL, the condition will eventually initiate an interrupt when the proper enabling conditions are met if the condition is still present.
- The previous mode field in the PSL is always set to kernel on an interrupt; but on an exception, it indicates the mode of the exception.

Processor errors, if not inconsistent with instruction completion, should create high priority interrupt requests. Otherwise, they must terminate instruction execution with an exception (fault, trap, or abort), in which case there may also be an associated interrupt request.

Error notification interrupts may be delayed from the apparent completion of the instruction in execution at the time of the error. But if enabled, the interrupt must be requested before processor context is switched, priority permitting.

An example of a case where both an interrupt and an exception are associated with the same event occurs when the VAX-11/780

instruction buffer gets an uncorrectable memory-read error. In this case, the interrupt request associated with the error will not be taken if the priority of the running program is high; but an abort will occur when an attempt is made to execute the instruction. The interrupt is still pending, however, and will be taken when the priority is lowered.

5.4 Processor Status

When an exception or an interrupt is serviced, the processor status must be preserved so that the interrupted process may continue normally. Basically, this is done by automatically saving the PC and the PSL on the stack. The PC and PSL are later restored with the Return from Exception or Interrupt (REI) instruction. Any other status required to correctly resume an interruptible instruction is stored in the general registers. The terms "current PSL" and "saved PSL" are used to distinguish between this status information when it is in the processor and when copies of it are materialized in memory, as on the stack.

Process context such as the mapping information is not saved or restored on each interrupt or exception. Instead, it is saved and restored only when process context switching is performed. Refer to the LDPCTX and SVPCTX instructions in Chapter 6. Other processor status is changed even less frequently; refer to the privileged register descriptions in Chapter 8.

5.5 Restartability

The VAX architecture requires that all instructions be restartable after a fault or interrupt that terminated execution before the instruction was completed. Generally, this means that modified registers are restored to the value they had at the start of execution, and no memory operands can be written. For some complex or iterative instructions, described in Chapter 3, intermediate results are stored in the general registers, and memory contents may have been altered. For most instructions with only a single modified or written operand, this implies special processing only when a multiple-byte operand spans a protection boundary, making it necessary to test accessibility of both parts of the operand. If the instruction has more than one operand to be written, every operand must be probed before any are written. (Writes to operands

in memory above the top of the stack are not constrained by this rule. See section 5.10.5.)

Instructions that store intermediate results in the general registers must not compromise system integrity. Therefore, they must ensure that any addresses stored or used are virtual addresses, subject to protection checking. In addition, any state information stored or used cannot result in a non-interruptible or non-terminating sequence.

Instruction operands that are peripheral-device registers having access side effects may produce UNPREDICTABLE results due to instruction restarting after faults or interrupts. For software to dependably access peripheral-device registers, instructions used to access them must not permit a fault or interrupt after the first I/O space access. See Chapter 7.

The setting of PTE⟨M⟩ is specifically excluded from the constraint that memory not be altered until the instruction can be completed.

5.6 *Interrupts*

The processor services interrupt requests between instructions. The processor also services interrupt requests at well-defined points during the execution of long, iterative instructions such as the string instructions. For these instructions, interrupts are initiated when the instruction state can be completely contained in the registers, PSL, and PC; saving additional instruction state in memory is thus avoided.

The following events cause interrupts:

- Device completion (IPL 10 through 17, hex)
- Device error (IPL 10 through 17, hex)
- Device alert (IPL 10 through 17, hex)
- Device memory error (IPL 10 through 17, hex)
- Console terminal transmit and receive (IPL 14, hex)
- Interval timer (implementation dependent, IPL 16 or 18, hex)
- Recovered memory or bus or processor errors (implementation dependent, IPL 18 through 1D, hex)
- Bus errors, processor errors, or uncorrectable memory errors (implementation dependent, IPL 18 through 1D, hex)

- Powerfail (IPL 1E, hex)
- Software interrupt invoked by *MTPR src, #PR$_SIRR* (IPL 01 through 0F, hex)
- AST delivery when REI restores a PSL with mode greater than or equal to ASTLVL (see Chapter 6) (IPL 02)

Each device controller has a separate set of interrupt vector locations in the system control block (SCB). Thus, interrupt service routines do not need to poll controllers to determine which controller interrupted.

To reduce interrupt overhead, no memory mapping information is changed when an interrupt occurs. Thus, the instructions, data, and contents of the interrupt vector for an interrupt service routine must be in the system address space or present in every process at the same address.

5.6.1 Urgent Interrupts

The processor provides eight priority levels (18 through 1F, hex) for use by urgent conditions, including serious errors and powerfail. Some implementations may not use all eight priority levels. Interrupts on these levels are initiated by the processor upon detection of certain conditions. Some of these conditions are not interrupts. For example, machine-check is usually an exception, but it runs at a high priority level on the interrupt stack.

Interrupt level 1E (hex) is reserved for powerfail. Interrupt level 1F (hex) is reserved for those exceptions that must lock out all processing until the condition has been handled. This includes the hardware and software "disasters" (machine-check and kernel-stack-not-valid aborts). It might also be used to allow a kernel-mode debugger to gain control on any exception.

5.6.2 Device Interrupts

The processor provides eight priority levels (10 through 17, hex) for use by peripheral devices. Some implementations may not implement all eight levels of interrupts. The minimal implementation is levels 14 through 17 (hex) that correspond to the UNIBUS levels BR4 through BR7 if the system has a UNIBUS interconnect.

5.6.3 Software Interrupts

The processor provides 15 interrupt levels (1 through 0F, hex) for use by software. Pending software interrupts are recorded in the

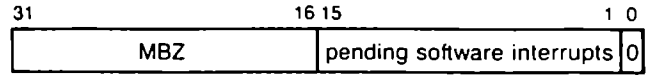


Figure 5.1
Software-Interrupt Summary Register

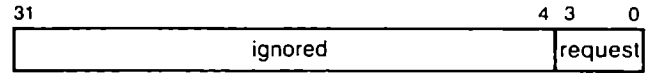


Figure 5.2
Software-Interrupt Request Register

software-interrupt-summary register (SISR), as shown in Figure 5.1. The SISR contains ones in the bit positions corresponding to levels at which software interrupts are pending. When the processor initiates a software interrupt, the corresponding bit in SISR is cleared. At no time can SISR contain ones in bits corresponding to levels higher than the current processor IPL, since the processor would already have taken the requested interrupts.

At processor initialization, SISR is cleared. The mechanism for accessing it follows:

MFPR #PRS_SISR, dst Reads SISR.

MTPR src, #PRS_SISR Loads SISR, but this is not the normal way of making software interrupt requests. It is useful, for example, for clearing the software interrupt system and for reloading its state during powerfail recovery.

The software-interrupt-request register (SIRR) is a write-only, 4-bit, privileged register used for creating software interrupt requests. SIRR is shown in Figure 5.2.

Executing *MTPR src, #PRS_SIRR* requests an interrupt at the level specified by src⟨3:0⟩. Once a software interrupt request is made, it will be cleared by the hardware when the interrupt is taken. If src⟨3:0⟩ is greater than the current IPL, the interrupt occurs before execution of the following instruction. If src⟨3:0⟩ is less than or equal to the current IPL, the interrupt will be deferred until IPL is lowered to less than src⟨3:0⟩ and there is no higher

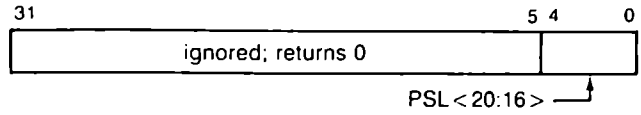


Figure 5.3
Interrupt-Priority-Level Register

interrupt level pending. This lowering of IPL is by either *REI* or *MTPR src, #PRS_IPL*. If src⟨3:0⟩ is 0, no interrupt will occur.

Note that no indication is given if there is already a request at the selected level. The service routine, therefore, must not assume that there is a one-to-one correspondence of interrupts generated and requests made. A valid protocol for generating such a correspondence is as follows:

1. The requester uses INSQUE to place a control block describing the request onto a queue for the service routine.
2. The requester uses *MTPR src, #PRS_SIRR* to request an interrupt at the appropriate level.
3. The service routine uses REMQUE to remove a control block from the queue of service requests. If REMQUE returns failure (nothing in the queue), the service routine exits with REI.
4. If REMQUE returns success (an item was removed from the queue), the service routine performs the service and returns to step 3 to look for other requests.

5.6.4 *Interrupt Priority Level Register*

Writing to the IPL register with the MTPR instruction will load the processor priority field in the PSL, that is, PSL⟨20:16⟩ is loaded from IPL⟨4:0⟩. Reading from the IPL register with the MFPR instruction will read the processor priority field from the PSL. On writing the IPL register, bits ⟨31:5⟩ are ignored; on reading the IPL register, bits ⟨31:5⟩ are returned 0. The IPL register is shown in Figure 5.3. At processor initialization, IPL is set to 1F (hex).

Interrupt service routines must follow the discipline of not lowering IPL below their initial level. If a service routine lowers IPL below its initial level, an interrupt could occur at an intermediate level, which will cause the stack nesting to be improper, which will result in REI faulting. If IPL is lowered to zero when the processor is running on the interrupt stack, the operation of the

Table 5.1
An Example of Interrupt Processing

Event	State after Event		
	IPL (hex)	SISR (hex)	Stacked PSL(IPL)
Initial state	5	00	0
Execute *MTPR #8, #PRS_IPL*	8	00	0
Execute *MTPR #3, #PRS_SIRR*	8	08	0
Execute *MTPR #7, #PRS_SIRR*	8	88	0
Execute *MTPR #9, #PRS_SIRR* (interrupts at once)	9	88	8,0
Device interrupt at IPL 20 (decimal)	14	88	9,8,0
Device interrupt service routine executes *REI*	9	88	8,0
IPL 9 service routine executes *REI*	8	88	0
Execute *MTPR #5, #PRS_IPL*[a]	7	08	5,0
IPL 7 service routine executes *REI*	5	08	0
Initial IPL 5 service routine executes *REI*[a]	3	00	0
IPL 3 service routine executes *REI*	0	00	—

[a]This operation lowers IPL below that of an outstanding software interrupt request. The software interrupt occurs at once.

processor is UNDEFINED. Table 5.1 is an example of interrupt processing.

5.7 Exceptions

Exceptions can be grouped into six classes:

- Arithmetic exceptions
- Memory management exceptions
- Exceptions detected during operand reference
- Exceptions occurring as a consequence of an instruction
- Tracing
- Serious system failures

5.7.1 Arithmetic Exceptions

Arithmetic exceptions may occur as the result of an arithmetic or conversion operation. These exceptions are mutually exclusive, and all are assigned the same vector in the SCB; hence the same signal "reason" code. Each of these exceptions indicate that an exception condition was detected during the last instruction and that the instruction has been completed (trap) or backed up (fault).

type code	:(SP)
PC of next instruction to execute	
PSL	

Figure 5.4
Arithmetic-Exception Stack Frame

Table 5.2
Arithmetic-Exception Type Codes

Exception Type	Mnemonic	Decimal	Hex
Traps			
integer overflow	SS$_INTOVF	1	1
integer divide-by-zero	SS$_INTDIV	2	2
decimal divide-by-zero	SS$_FLTDIV	4	4
decimal overflow	SS$_DECOVF	6	6
subscript range	SS$_SUBRNG	7	7
Faults			
floating overflow	SS$_FLTOVF_F	8	8
floating divide-by-zero	SS$_FLTDIV_F	9	9
floating underflow	SS$_FLTUND_F	10	A

An appropriate distinguishing code is pushed on the stack as a longword, as shown in Figure 5.4. Table 5.2 lists the arithmetic-exception type codes.

Integer Overflow Trap. An integer overflow trap is an exception that indicates that the last instruction executed had an integer overflow setting PSL⟨V⟩ and that integer overflow was enabled (PSL⟨IV⟩ was set). The result stored is the low-order part of the correct result. PSL⟨N⟩ and PSL⟨Z⟩ are set according to the stored result. The type code pushed on the stack is 1 (SS$_INTOVF).

Integer Divide-by-Zero Trap. An integer divide-by-zero trap is an exception that indicates that the last instruction executed had an integer zero divisor. The result stored is equal to the dividend, and PSL⟨V⟩ is set. The type code pushed on the stack is 2 (SS$_INTDIV).

Divide-by-Zero Trap. A decimal string divide-by-zero trap is an exception that indicates that the last instruction executed had a decimal-string zero divisor. The destination, R0 through R5, and condition codes are UNPREDICTABLE. The zero divisor can be either +0 or –0. The type code pushed on the stack is 4 (SS$_FLTDIV).

Decimal-String Overflow Trap. A decimal-string overflow trap is an exception that indicates that the last instruction executed had a decimal-string result too large for the destination string provided and that decimal overflow was enabled (PSL⟨DV⟩ was set). The PSL⟨V⟩ condition code is always set. Refer to the individual instruction descriptions in Chapter 3 for the value of the result and of the condition codes. The type code pushed on the stack is 6 (SS$_DECOVF).

Subscript-Range Trap. A subscript-range trap is an exception that indicates that the last instruction was an INDEX instruction with a subscript operand that failed the range check. The value of the subscript operand is lower than the low operand or greater than the high operand. The result is stored in the indexout operand, and the condition codes are set as if the subscript were within range. The type code pushed on the stack is 7 (SS$_SUBRNG).

Floating Overflow Fault. A floating overflow fault is an exception that indicates that the last instruction executed resulted in an exponent greater than the largest representable exponent for the data type after normalization and rounding. The destination was unaffected, and the saved condition codes are UNPREDICTABLE. The saved PC points to the instruction causing the fault. In the case of a POLY instruction, the instruction is suspended with PSL⟨FPD⟩ set. The type code pushed on the stack is 8 (SS$_FLTOVF_F).

Floating Divide-by-Zero Fault. A floating divide-by-zero fault is an exception that indicates that the last instruction executed had a floating zero divisor. The quotient operand was unaffected, and the saved condition codes are UNPREDICTABLE. The saved PC points to the instruction causing the fault. The type code pushed on the stack is 9 (SS$_FLTDIV_F).

Floating Underflow Fault. A floating underflow fault is an exception that indicates that the last instruction executed resulted in an exponent less than the smallest representable exponent for the data type after normalization and rounding, and that floating underflow was enabled (PSL⟨FU⟩ was set). The destination operand is unaffected. The saved condition codes are UNPREDICTABLE. The saved PC points to the instruction causing the fault. In the case of a POLY instruction, the instruction is suspended with PSL⟨FPD⟩ set. The type code pushed on the stack is 10 (SS$_FLTUND_F).

Note

Floating overflow, floating underflow, and floating divide by zero were originally implemented as traps on the VAX-11/780, and had type codes 3, 4, and 5, respectively. The architecture was later modified to include only floating-point faults, and all VAX-11/780s were upgraded.

5.7.2 *Memory Management Exceptions*

A memory management exception can be an access-control-violation fault, a translation-not-valid fault, or a modify fault. See Chapter 4 for a description of the information pushed on the stack as parameters by the memory management exceptions.

Note that in a reference to a single page, if two or more memory management exceptions are indicated, access-control-violation takes precedence over both translation-not-valid and modify fault.

Access-Control-Violation Fault. An access-control-violation fault is an exception indicating that the process attempted a reference not allowed at the current access mode. Software may restart the process after changing the address translation information.

Translation-Not-Valid Fault. A translation-not-valid fault is an exception indicating that the process attempted a reference to a page for which the valid bit in the page-table entry was not set. Software may restart the process after making the page valid.

Modify Fault. A modify fault is an exception indicating that the process attempted a write-access or modify-access reference to a page for which the modify bit in the page-table entry was not set. The processor flushes the faulting PTE from the TB as part

of the process of initiating a modify fault. Software may restart the process after setting PTE⟨M⟩. Not all processors take modify faults; some implementations set PTE⟨M⟩ and continue.

5.7.3 Exceptions Detected During Operand Reference

Reserved-Addressing-Mode Fault. A reserved-addressing-mode fault is an exception indicating that an operand specifier included an addressing mode that is not allowed in the situation in which it occurred. No parameters are pushed.

See Chapter 2 for details of reserved addressing modes and for combinations of addressing modes and registers that cause UNPREDICTABLE results.

Reserved-Operand Exception. A reserved-operand exception is an exception indicating that an operand accessed has a format reserved for future use by DIGITAL. No parameters are pushed. This exception always backs up the saved PC to point to the opcode. The exception service routine may determine the type of operand by examining the opcode using the saved PC.

Note that only the changes made by instruction fetch and operand specifier evaluation may be restored. Therefore, some instructions are not restartable. These exceptions are labeled as aborts rather than faults. The saved PC is always restored properly unless the instruction attempted to modify it in a manner that results in UNPREDICTABLE results.

The reserved-operand exceptions are caused by the following conditions:

- Bit field too wide (fault)
- Invalid combination of bits in PSL restored by REI (fault)
- Invalid combination of bits in PSW mask longword during RET (fault)
- Invalid combination of bits in BISPSW or BICPSW (fault)
- Invalid CALLS or CALLG entry mask (fault)
- Invalid register number in MFPR or MTPR (fault)
- Invalid PCB contents in LDPCTX for some implementations (abort)
- Unaligned operand in ADAWI (fault)

- Invalid register contents in MTPR for some implementations (fault)
- Invalid operand addresses in INSQHI, INSQTI, REMQHI, or REMQTI (fault)
- A floating-point number that has the sign bit set and the exponent zero except in the POLY table (fault)
- A floating-point number that has the sign bit set and the exponent zero in the POLY table (fault)
- POLY degree too large
- Decimal string too large (abort)
- Invalid digit in CVTTP or CVTSP (abort)
- Reserved pattern operator in EDITPC (fault)
- Incorrect source-string length at completion of EDITPC (abort)

5.7.4 *Exceptions Occurring as the Consequence of an Instruction*

Reserved-Instruction Fault. A reserved-instruction fault occurs when the processor encounters an opcode that is reserved for future use by DIGITAL. No parameters are pushed. Opcode FFFF (hex) is reserved for all time.

Privileged-Instruction Fault. A privileged-instruction fault occurs when the processor encounters an instruction that requires higher privileges than the current processor mode. No parameters are pushed. Privileged-instruction fault and reserved-instruction fault use the same SCB vector, and cannot be distinguished without examining the faulting instruction.

Opcode Reserved-to-Customers Fault. An opcode-reserved-to-customers fault is an exception that occurs when an opcode reserved to customers is executed. The operation is identical to the reserved-instruction fault except that the event is caused by a different set of opcodes and faults through a different vector. All opcodes reserved to customers start with FC (hex), which is the XFC instruction. If the special instruction needs to generate a unique exception, one of the reserved-to-customer vectors should be used. An example might be an unrecognized second byte of the instruction.

The XFC fault is intended primarily for use with writable control store to implement installation-dependent instructions. The method used to enable and disable the handling of an XFC fault in

user-written microcode is implementation dependent. Some implementations may transfer control to microcode without checking bits ⟨1:0⟩ of the exception vector.

Instruction-Emulation Exceptions. When a processor executes a string instruction that is omitted from its instruction set, an emulation exception results. An emulation exception can occur through either of two SCB vectors, depending on whether PSL⟨FPD⟩ was set at the beginning of the instruction. If PSL⟨FPD⟩ is clear, an instruction-emulation trap occurs through the SCB vector at offset C8 (hex), and a instruction-emulation trap frame is pushed on the current stack. If PSL⟨FPD⟩ is set, a suspended-emulation fault occurs through the SCB vector at offset CC (hex) and PC and PSL are pushed on the current stack.

Before any information is pushed on the stack, the stack is probed. If the probe faults, the instruction then faults, or otherwise the exception continues as described above.

The emulation exception handler runs in the mode of the emulated instruction, on the same stack, and at the same IPL. The exception parameters are pushed onto the current stack. See Chapter 11 for details of instruction emulation and related emulation exceptions.

Compatibility-Mode Exceptions. A compatibility-mode exception is an exception that occurs when the processor is in compatibility mode. A longword of information is pushed on the stack, which contains a code indicating the exception type. The stack frame is the same as that for arithmetic exceptions (shown in Figure 5.4). The compatibility mode exception type codes are shown in Table 5.3.

All other exceptions in compatibility mode occur to the regular native-mode vector; for example, access-control-violation fault, translation-not-valid fault, and machine-check abort (see Chapter 9).

Change-Mode Trap. A change-mode trap is an exception that occurs when one of the change-mode instructions (CHMK, CHME, CHMS, CHMU) is executed. The instruction operand is pushed

Table 5.3
Compatibility-Mode Exception Type Codes

Exception Type	Code
Faults	
reserved opcode	0
BPT instruction	1
IOT instruction	2
EMT instruction	3
TRAP instruction	4
illegal instruction	5
Aborts	
odd address	6

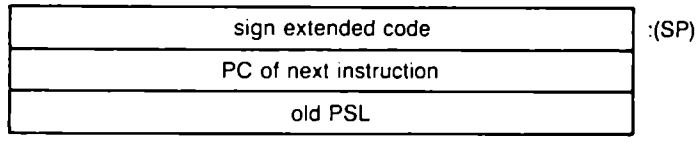


Figure 5.5
CHMx Stack Frame

on the exception stack, as shown in Figure 5.5. See the description of the change-mode instructions for details.

Breakpoint Fault. A breakpoint fault is an exception that occurs when the breakpoint instruction (BPT) is executed. No parameters are pushed.

To proceed from a breakpoint, a debugger or tracing program typically restores the original contents of the location containing the BPT, sets T in the PSL saved by the BPT fault, and resumes. When the instruction completes, a trace exception will occur (see section 5.7.5). At this point, the tracing program can again re-insert the BPT instruction, restore PSL⟨T⟩ to its original state (usually clear), and resume. Note that if both tracing and breakpointing are in progress (if PSL⟨T⟩ was set at the time of the BPT), then on

the trace exception both the BPT restoration and a normal trace exception should be processed by the trace handler.

5.7.5 *Trace Fault*

A trace is an exception that occurs between instructions when trace is enabled. Tracing is used for tracing programs, for performance evaluation, and for debugging purposes. It is designed so that one and only one trace exception occurs before the execution of each traced instruction. The saved PC on a trace is the address of the next instruction that normally would be executed. If a trace fault and a memory management fault (or an odd-address abort during a compatibility-mode instruction fetch) occur simultaneously, the order in which the exceptions are taken is UNPREDICTABLE. The trace fault for an instruction takes precedence over all other exceptions.

To ensure that exactly one trace occurs per instruction despite other traps and faults, the PSL contains two bits: trace enable (T) and trace pending (TP). If only one bit were used, the occurrence of an interrupt at the end of an instruction would produce either zero or two traces, depending on the design. Instead of the PSL⟨T⟩ bit being defined to produce a trap after any other traps or aborts at the end of an instruction, the trap effect is implemented by copying PSL⟨T⟩ to a second bit (PSL⟨TP⟩) that is actually used to generate the exception. PSL⟨TP⟩ generates a fault before any other processing at the start of the next instruction.

The rules of operation for trace are as follows:

1. At the beginning of an instruction, if PSL⟨TP⟩ is set, a trace fault is taken after clearing PSL⟨TP⟩.
2. PSL⟨TP⟩ is loaded with the value of PSL⟨T⟩.
3. If the instruction faults or an interrupt is serviced, PSL⟨TP⟩ is cleared before the PSL is saved on the stack. The saved PC is set to the start of the faulting or interrupted instruction. Instruction execution is resumed at step 1.
4. If the instruction aborts or takes an arithmetic trap, PSL⟨TP⟩ is not changed before the PSL is saved on the stack.
5. If an interrupt is serviced after instruction completion and arithmetic traps but before tracing is checked for at the start of the next instruction, then PSL⟨TP⟩ is not changed before the PSL is saved on the stack.

The routine entered by a CHMx is not traced because CHMx clears T and TP in the new PSL. However, if PSL⟨T⟩ was set at the beginning of CHMx, the saved PSL will have both T and TP set. Trace faults resume with the instruction following the REI in the routine entered by the CHMx. An instruction following an REI will fault either if PSL⟨T⟩ was set when the REI was executed or if TP in the saved PSL is set; in both cases, PSL⟨TP⟩ is set after the REI. Note that a trace fault that occurs for an instruction following an REI that sets PSL⟨TP⟩ will be taken with the new PSL. Thus, special care must be taken if exception or interrupt routines are traced. If PSL⟨T⟩ is set by a BISPSW instruction, trace faults begin with the second instruction after the BISPSW.

In addition, the CALLS and CALLG instructions save a clear T, although T in the PSL is unchanged. This is done so that a debugger or trace program proceeding from a BPT fault does not get a spurious trace from the RET that matches the CALL.

The detection of reserved-instruction faults occurs after the trace fault. The detection of interrupts and other exceptions can occur during instruction execution. In this case, PSL⟨TP⟩ is cleared before the exception or interrupt is initiated. The entire PSL (including T and TP) is automatically saved on interrupt or exception initiation and is restored at the end with an REI. This makes interrupts and benign exceptions totally transparent to the executing program.

Table 5.4 shows the operation of tracing during execution of ordinary instructions, instructions that have special effects on tracing, and other system events that effect tracing.

Routines using the trace facility are termed "trace handlers." They should observe the following conventions and restrictions:

1. When the trace handler performs its REI back to the traced program, it should always force the T-bit to be set in the PSL that will be restored. This defends against programs clearing PSL⟨T⟩ with RET, REI, or BICPSW.
2. The trace handler should never examine or alter PSL⟨TP⟩ when continuing tracing. The hardware flows ensure that this bit is maintained correctly to continue tracing.

Table 5.4
State Changes During Tracing

Instruction or Instruction Sequence	Original PSL⟨T⟩	Final PSL⟨T⟩	Final PSL⟨TP⟩	Comments
Ordinary instruction	0	0	0	
	1	1	1	
CHMx, through the REI that ends the service routine	0	0	0	
	1	1	1	
Interrupt or exception, through the REI that ends the service routine	0	0	0	
	1	1	1	
CALLS or CALLG	0	0	0	
	1	1	1	(pushed PSW⟨T⟩ clear)
RET	0	*	0	
	1	*	1	
CHMx	0	0	0	(pushed PSL⟨TP⟩ clear)
	1	0	0	(pushed PSL⟨TP⟩ set)
REI, when the stacked PSL⟨TP⟩ is clear	0	*	0	
	1	*	1	
REI, when the stacked PSL⟨TP⟩ is set	0	*	1	
	1	*	1	
BISPSW that sets PSW⟨T⟩	0	1	0	
	1	1	1	
BICPSW that clears PSW⟨T⟩	0	0	0	
	1	0	1	
interrupt or exception	0	0	0	(pushed PSL⟨TP⟩ clear)
	1	0	0	(pushed PSL⟨TP⟩ depends on above description)

* Depends on PSW⟨T⟩ popped from the stack.

3. When tracing is to be ended, both PSL⟨T⟩ and PSL⟨TP⟩ should be cleared. This ensures that no further traces will occur.
4. Tracing a service routine that completes with an REI will give a trace in the restored mode after the REI. If the program being restored to was also being traced, only one trace exception is generated.
5. If a routine entered by a CALLS or CALLG instruction is executed at full speed by turning off T, trace control can be

regained by setting T in the PSW in its call frame. Tracing will resume after the instruction following the RET.

6. Tracing is disabled for routines entered by a CHMx instruction or any exception. Thus, if a CHMx or exception service routine is to be traced, a breakpoint instruction must be placed at its entry point. If such a routine is recursive, breakpointing will catch each recursion only if the breakpoint is not on the CHMx or instruction with the exception.
7. If it is desired to allow multiple trace handlers, all handlers should preserve PSL⟨T⟩ when turning on and off trace. They also would have to simulate traced code that alters or reads PSL⟨T⟩.

5.7.6 *Serious System Failures*

Kernel-stack-not-valid abort is an exception that indicates that a memory management problem occurred while the processor was pushing information on the kernel stack during the initiation of an exception or interrupt. Usually this is an indication of a stack overflow or other operating system error. The attempted exception is transformed into an abort that uses the interrupt stack. No extra information is pushed on the interrupt stack in addition to PSL and PC of the original exception. IPL is raised to 1F (hex). If the exception vector ⟨1:0⟩ is not 1, the operation of the processor is UNDEFINED.

Software may abort the process without aborting the system. Because of the lost information, however, the process cannot be continued. If a memory management problem occurs on a normal instruction reference to the kernel stack (as with CHMx or REI), the normal memory management fault is initiated.

An interrupt-stack-not-valid halt results when a memory management problem a memory error occurs while the processor is pushing information on the interrupt stack during the initiation of an exception or interrupt. No further interrupt requests are acknowledged on the processor. The processor leaves the PC, the PSL, and the reason for the halt in registers so that they are available to a debugger, to the normal bootstrap routine, or to an optional watchdog bootstrap routine. A watchdog bootstrap can cause the processor to leave the halted state.

Software should ensure that the kernel and interrupt stacks are

00000010 (hex)	:(SP)
1st longword of error report	
2nd longword of error report	
3rd longword of error report	
4th longword of error report	
PC	
PSL	

Figure 5.6
An Example Machine-Check Stack Frame

mapped valid, writable (at least to kernel mode), and modified, so as to avoid kernel-stack-not-valid aborts and interrupt-stack-not-valid halts.

A machine-check exception indicates that the processor detected an internal error. As is usual for exceptions, machine-check is taken regardless of current IPL. The machine-check exception vector bits⟨1:0⟩ must specify 1, or the operation of the processor is UNDEFINED. The exception is taken on the interrupt stack, and IPL is raised to 1F (hex).

The processor pushes a machine-check stack frame on the interrupt stack, consisting of a count longword, an implementation-dependent number of error report longwords, and a PC and PSL. The count longword reports the number of bytes of error report pushed. For example, if 4 longwords of error report are pushed, the count longword will contain 16 (decimal). An example machine-check stack frame is shown in Figure 5.6.

Software can decide, on the basis of the information presented, whether to abort the current process if the machine-check came from the process. Machine-check includes uncorrected bus and memory errors anywhere and any other processor-detected errors. Some processor errors cannot ensure the state of the machine at all. For such errors, the state will be preserved on a "best effort" basis.

5.8 *Serialization of Notification of Multiple Events*

The interaction between arithmetic traps, tracing, other exceptions, and multiple interrupts is complex. To ensure consistent and useful implementations, it is necessary to understand this interaction at a detailed level. As an example, if an instruction is started with PSL⟨T⟩ = 1 and PSL⟨TP⟩ = 0, it gets an arithmetic trap and an interrupt request is recognized, then the following sequence of events occurs:

1. The instruction finishes, storing all its results. PSL⟨TP⟩ is set at the end of this instruction since PSL⟨T⟩ was set at the beginning.
2. The overflow trap sequence is initiated, saving PC and PSL (with TP=1), loading a new PC from the overflow trap vector, and creating a new PSL.
3. The interrupt sequence is initiated, saving the PC and PSL appropriate to the overflow-trap service routine, loading a new PC from the interrupt vector, and creating a new PSL.
4. If a higher priority interrupt is noticed, the first instruction of the interrupt service routine is not executed. Instead, the PC and PSL appropriate to that routine are saved as part of initiating the new interrupt. The original interrupt service routine will then be executed when the higher priority routine terminates with REI.
5. The interrupt service routine runs and then exits with REI.
6. The overflow-trap service routine runs and then exits with REI, which sets PSL⟨TP⟩ since the saved PSL⟨TP⟩ was set.
7. The trace fault occurs, again pushing PC and PSL but this time with PSL⟨TP⟩=0.
8. Trace service routine runs and then exits with REI.
9. The next instruction is executed.

This sequence is accomplished by the following operation between instructions:

```
! Here at completion of instruction, including
!  at end of REI from an exception or interrupt routine.

1$:  possibly take interrupts or console halt;
! If so, PSL<TP> is not modified before PSL is saved.

if PSL<TP> EQLU 1 then     ! If trace pending, then fault.
   begin                   ! Trace fault takes precedence
   PSL<TP> <- 0;           !  over other exceptions.
```

```
   initiate trace fault;
   end;

possibly take interrupts or console halt;
! If so, PSL<TP> is not modified before PSL is saved.

PSL<TP> <- PSL<T>;       ! If trace enable, set trace pending

go start instruction execution;
! Reserved instruction faults are taken here.
! PSL<FPD> is tested here, thus PSL<TP> takes
!  precedence over PSL<FPD> if both are set.
if instruction faults OR an interrupt or console halt
   is taken before end of instruction then
   begin
   back up PC to start of opcode;
   either set PSL<FPD> or back up all general
      register side effects;
   PSL<TP> <- 0;
   initiate exception or interrupt;
   end;

if arithmetic trap needed and no other abort or trap
   then initiate arithmetic trap;

! Note: All instructions end by flowing
!  through 1$, thus the REI from a service
!  routine will return to 1$.
```

5.9 System Control Block

The system control block is a data structure in memory, containing the vectors by which exceptions and interrupts are dispatched to the appropriate service routines. Table 5.5 shows the interrupt and exception vectors in the SCB.

5.9.1 System Control Block Base

The system control-block base (SCBB) is a privileged register containing the address of the system control block, which must be page-aligned. Figure 5.7 shows the SCBB. SCBB may contain either a virtual or a physical address, depending on the processor implementation.

For a processor with a 24-bit, 30-bit, 32-bit one-to-one, or 32-bit sign-extended physical address mode, the processor locates the SCB in physical memory and holds a physical address in SCBB.

Table 5.5
System Control Block Vectors (Part 1)

Offset	Name	Type	Parameters	Notes
00	passive release	interrupt		IPL is that of the request.
04	machine check	exception	varies	Number of parameters is implementation dependent.
08	kernel stack not valid	abort	0	
0C	power fail	interrupt		IPL is 1E.
10	reserved or privileged instruction	fault	0	Opcodes reserved to DIGITAL and privileged instructions.
14	customer reserved instruction	fault	0	XFC instruction.
18	reserved operand	fault or abort	0	
1C	reserved addressing mode	fault	0	
20	access-control violation	fault	2	Virtual address and fault parameter are pushed.
24	translation not valid	fault	2	Virtual address and fault parameter are pushed.
28	trace pending	fault	0	
2C	breakpoint instruction	fault	0	
30	compatibility	fault or abort	1	A type code is pushed.
34	arithmetic	trap or fault	1	A type code is pushed.
38	reserved			

For a processor with a 34-bit physical address mode, the processor may locate the SCB in either physical or virtual memory and holds a physical or virtual address in SCBB, respectively.

The entire SCB must reside in the portion of physical memory that is accessible by virtual address translation when memory management is off. For example, on a processor that translates a virtual address to a 34-bit physical address by the preferred method described in Section 4.2.1, software must ensure that the SCB resides within the first 512 megabytes of physical memory.

Processor initialization leaves the contents of SCBB UNPREDICTABLE.

If the SCBB points to I/O space or nonexistent memory when

Table 5.5
System Control Block Vectors (Part 2)

Offset	Name	Type	Parameters	Notes
3C	modify	fault	2	Virtual address and fault parameter are pushed.
40	CHMK	trap	1	The operand word is sign extended and pushed.
44	CHME	trap	1	The operand word is sign extended and pushed.
48	CHMS	trap	1	The operand word is sign extended and pushed.
4C	CHMU	trap	1	The operand word is sign extended and pushed.
50–60	reserved for bus or memory error	interrupt		IPL is implementation dependent.
64	unused			Reserved to DIGITAL.
68	vector processor disabled	fault	0	See Chapter 13 for details
6C–7C	unused			Reserved to DIGITAL.
80	interprocessor interrupt	interrupt		IPL is 16 (hex).
84	software level 1	interrupt		IPL is 1.
88	software level 2	interrupt		IPL is 2. Ordinarily used for AST delivery.
8C	software level 3	interrupt		IPL is 3. Ordinarily used for process scheduling.
90–BC	software levels 4–F	interrupt		Vector corresponds to IPL.
C0	interval timer	interrupt		IPL is 16 or 18 (hex), implementation dependent.
C4	unused			Reserved to DIGITAL.

an exception or interrupt occurs, the operation of the processor is UNDEFINED. If SCBB contains a virtual address, the page or pages containing the SCB must be valid and accessible to kernel mode, or the operation of the processor is UNDEFINED.

5.9.2 *Interrupt and Exception Vectors*

An interrupt or exception vector is a longword in the SCB. The processor examines the vector when an exception or interrupt occurs to determine how to service the event.

Separate vectors are defined for each interrupting device controller

Table 5.5
System Control Block Vectors (Part 3)

Offset	Name	Type	Parameters	Notes
C8	instruction emulation	trap	10	FPD clear. Emulation frame is pushed.
CC	suspended emulation	fault	0	FPD set.
D0 - DC	unused	Reserved to DIGITAL		
E0 - EC	unused	Reserved to customers		
F0	console storage receive	interrupt		11/750 and 11/730. IPL is implementation dependent.
F4	console storage transmit	interrupt		11/750 and 11/730. IPL is implementation dependent.
F8	console terminal receive	interrupt		IPL is 14 (hex).
FC	console terminal transmit	interrupt		IPL is 14 (hex).
100–13C	adapter vectors	interrupt		IPL is 14 (hex). Implementation dependent.
140–17C	adapter vectors	interrupt		IPL is 15 (hex). Implementation dependent.
180–1BC	adapter vectors	interrupt		IPL is 16 (hex). Implementation dependent.
1C0–1FC	adapter vectors	interrupt		IPL is 17 (hex). Implementation dependent.
200–3FC	device vectors	interrupt		Implementation dependent.
400–5FC	device vectors	interrupt		Implementation dependent.

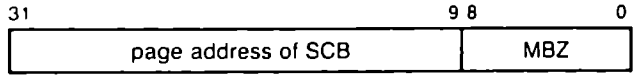


Figure 5.7
System Control-Block Base

and each class of exceptions. Each vector is interpreted as follows by the hardware. Bits ⟨1:0⟩ contain a code interpreted as follows:

0: Service this event on the kernel stack unless already running on the interrupt stack, in which case service on the interrupt stack.

1: Service this event on the interrupt stack. If this event is an exception, the IPL is raised to 1F (hex).

2: Service this event in writable control store, passing bits ⟨15:2⟩ to the installation-dependent microcode there. If writable control store does not exist or is not loaded, the operation is UNDEFINED.

3: Operation UNDEFINED. Reserved to DIGITAL.

For codes 0 and 1, bits ⟨31:2⟩ contain the virtual address of the service routine, which must begin on a longword boundary and which ordinarily will be in the system space. CHMx is serviced on the stack selected by the new mode. Bits ⟨1:0⟩ in the CHMx vectors must be zero, or the operation of the processor is UNDEFINED. Emulation exceptions are serviced on the current stack. Bits ⟨1:0⟩ in the emulation exception vectors must be zero, or the operation of the processor is UNDEFINED.

The assignment of SCB offsets and priority levels for controllers, adapters, and other devices connecting to the system bus is implementation dependent. Typically, interrupt priority levels 14 through 17 (hex) are used to signal I/O device, controller, and adapter events. Typically, one interrupt vector is assigned to each priority level for each adapter.

The use of second or third SCB pages (offsets 200 through 3FC and 400 through 5FC) is implementation dependent. In some processors (the VAX-11/750 and VAX-11/730, for example), UNIBUS devices interrupt the processor directly, and the second SCB page contains the UNIBUS device vectors. When a UNIBUS device connected to such a system requests an interrupt, the vector is determined by adding 200 (hex) to the vector supplied by the device. If a second UNIBUS adapter is installed, the third SCB page contains its device vectors, and 400 (hex) is added to the vector supplied by the device attached to the second UNIBUS interconnect. Only device vectors in the range 0 through 1FC (hex) are allowed. Interrupt priority levels 14 through 17 (hex) correspond to UNIBUS levels BR4 through BR7 if the system has a UNIBUS interconnect.

5.10 Stacks

At any time, the processor is in either: a process context (and PSL⟨IS⟩ = 0) in one of four modes (kernel, executive, supervisor,

user); or the system-wide interrupt service context (and PSL⟨IS⟩ = 1), which operates with kernel privileges. There is a stack pointer copy associated with each of these five states, and an internal processor register (IPR) to hold each stack pointer copy. Any time the processor changes from one of these states to another, SP (R14) is stored in the copy associated with the old state and SP is loaded from the copy associated with the new state. The five stack pointers, which are accessible as internal processor registers, are as follows:

KSP	Kernel-mode stack pointer
ESP	Executive-mode stack pointer
SSP	Supervisor-mode stack pointer
USP	User-mode stack pointer
ISP	Interrupt stack pointer

Operating system design must choose a priority level that is the boundary between kernel and interrupt stack use. The SCB interrupt vectors must be set such that interrupts to levels above this boundary run on the interrupt stack (vector⟨1:0⟩ = 1) and interrupts below this boundary run on the kernel stack (vector⟨1:0⟩ = 0). Typically, AST delivery (IPL 2) is on the kernel stack, and all higher levels are on the interrupt stack.

5.10.1 *Stack Residency*

The user-, supervisor-, and executive-mode stacks do not need to be resident. Kernel-mode code can bring in or allocate process stack pages as translation-not-valid faults occur. The kernel stack for the current process and the interrupt stack (which is process independent), however, must be resident, accessible, and modified. Translation-not-valid, access-control-violation, and modify faults occurring on references to either of these stacks are regarded as serious system failures. (Because the console may use 8 bytes of interrupt-stack space when starting a processor, the 8 bytes above the top of the interrupt stack, –8(SP) through –1(SP), must also be resident.)

If any of these faults occur while pushing exception or interrupt state on the interrupt stack, the processor halts. If a translation-not-valid or access-control-violation fault occurs while pushing an exception or interrupt state on the kernel stack, the processor aborts the current sequence and initiates kernel-stack-not-valid abort on hardware level 1F (hex). If modify fault occurs while

pushing an exception or interrupt state on the kernel stack, the processor is allowed to do one of two things: (1) set the PTE⟨M⟩ bit for that page and continue normal operation; or (2) generate a kernel-stack-not-valid abort on hardware level 1F (hex). The processor may implement the first option even if it supports modify fault.

The behavior just described does not mean that every possible reference to the kernel and interrupt stacks is checked, but rather that the processor will not loop on these faults.

It is not necessary that the kernel stack for a process other than the current one be resident, but it must be resident before that process is selected to run by the software's process dispatcher. Further, any mechanism that uses memory management faults to gather process statistics, for instance, must exercise care not to fault kernel-stack or interrupt-stack pages.

5.10.2 *Stack Alignment*

Except on CALLS and CALLG instructions, the hardware makes no attempt to align the stacks. For best performance on all processors, the software should align the stack on a longword boundary and allocate the stack in longword increments. The convert-byte-to-long (CVTBL and MOVZBL), convert-word-to-long (CVTWL and MOVZWL), convert-long-to-byte (CVTLB), and convert-long-to-word (CVTLW) instructions are recommended as the methods for pushing bytes and words on the stack and popping them off to keep it longword aligned.

5.10.3 *Stack Status Bits*

The interrupt stack bit and current-mode bits in the PSL specify which of the five stack pointers is currently in use, as shown in Table 5.6.

The processor does not allow current mode to be non-zero when IS=1. This is achieved by clearing the mode bits when taking an interrupt or exception and by causing reserved operand fault if REI attempts to load a PSL in which both IS and mode are non-zero.

The stack to be used for an interrupt or exception is selected by the current PSL⟨IS⟩ and bits⟨1:0⟩ of the vector. If the current PSL⟨IS⟩ is 1 or if the low bits of the vector are 01 (binary), the

Table 5.6
Indication of Current Stack Pointer

Stack Pointer	Mnemonic	PSL⟨IS⟩	PSL⟨CUR_MOD⟩
Interrupt stack pointer	ISP	1	0
Kernel stack pointer	KSP	0	0
Executive stack pointer	ESP	0	1
Supervisor stack pointer	SSP	0	2
User stack pointer	USP	0	3

interrupt stack is used. If the current PSL⟨IS⟩ is 0 and the low bits of the vector are 00 (binary), then the kernel stack is used. Values 10 (binary) and 11 (binary) of the vector⟨1:0⟩ are used for other purposes. Refer to section 5.9 for details.

5.10.4 *Accessing Stack Registers*

Reference to SP (the stack pointer) in the general registers will access one of five possible architecturally defined stack pointers—the user, supervisor, executive, kernel, or interrupt—depending on the values of the current mode and IS bits in the PSL. Some processors may implement these five stack pointers as five internal processor registers. Other processors may store the four per-process stack pointers in memory in the PCB and store only the interrupt stack pointer in an internal register (see Chapter 6). In either case, software can access any of the five stack pointers with the MTPR and MFPR instructions. Results are correct even if the stack pointer specified by the current mode and IS bits in the PSL is referenced in the internal processor register space by an MTPR or MFPR instruction.

If the four process stack pointers are implemented as registers, then these instructions are the only method for accessing the stack pointers of the current process. See Chapter 8 for conventions to be followed when referencing other per-process registers in the internal processor register space.

The internal processor register numbers were chosen to be the same as PSL⟨26:24⟩. The previous stack pointer is the same as PSL⟨23:22⟩ unless PSL⟨IS⟩ is set. If PSL⟨IS⟩ is set, the previous mode cannot be determined from the PSL since interrupts always

clear PSL(23:22). Processor initialization leaves the contents of all stack pointers UNPREDICTABLE.

5.10.5 *Memory Above Top-of-Stack*

To ensure correct operation, software must assume that preceding the initiation of every instruction a processor will set the contents of all writable bytes between −1(SP) and −512(SP) to UNPREDICTABLE values. Bytes that cannot be written because of access-control violation, translation-not-valid, or modify fault are unaffected. For example:

```
MOVW    register_mask, -(SP)
PUSHR   (SP)+                   ; Predictable results

MOVW    register_mask, -2(SP)
PUSHR   -2(SP)                  ; UNPREDICTABLE results

MOVAB   arg_list, 10(SP)
CALLG   @10(SP), dst            ; Predictable results

MOVAB   arg_list, -10(SP)
CALLG   @-10(SP), dst           ; UNPREDICTABLE results

CALLG   (SP)+, @(SP)+           ; Predictable results
```

A VAX processor may take advantage of this in interpreting instructions that push operands on the stack by beginning to write the operands above the top of the stack without first having probed to ensure that all operands can be written. A VAX processor is not generally allowed to write memory before ensuring that the instruction will complete, but writes to operands above the top of the stack are explicitly excluded from this rule.

5.11 *Initiate Exception or Interrupt*

The following flow describes how exceptions and interrupts are initiated by a VAX processor. The flow begins at the point where the processor has determined the proper SCB vector to read.

Initiate Initiate Exception or Interrupt

Operation

```
! Read the SCB vector into a temporary register, and check it
! for validity.  The vector number is determined by the
! exception or interrupt type.
{disable interrupts};
vector ← SCB[vector_number];
case vector<1:0> of
   0: if {machine check OR kernel-stack-not-valid} then
         {UNDEFINED};
   1: if {CHMx OR instruction emulation exception} then
         {UNDEFINED};
   2: if {user microcode exists and is loaded} then
         {enter user microcode}
      else
         {UNDEFINED};
   3: {UNDEFINED};
end;

! Save the current PSL in a temporary register.
saved_PSL ← PSL;

! Create and load a new PSL.
case {exception or interrupt type} of
   {interrupt}:
      begin
      PSL<CM,TP,FPD,DV,FU,IV,T,N,Z,V,C> ← 0;
      PSL<CUR_MOD,PRV_MOD> ← 0;
      if vector <1:0> EQLU 1
         then PSL<IS> ← 1
         else PSL<IS> ← saved_PSL<IS>;
      PSL<IPL> ← new_IPL;
      end;
   {CHMx}:
      begin
      PSL<CM,TP,FPD,DV,FU,IV,T,N,Z,V,C> ← 0;
      PSL<CUR_MOD> ← new_mode;
      PSL<PRV_MOD> ← saved_PSL<CUR_MOD>;
      PSL<IS> ← saved_PSL<IS>;
      PSL<IPL> ← saved_PSL<IPL>;
      end;
```

```
   {instruction emulation exception}:
      begin
      PSL<CM,TP,FPD,DV,FU,IV,T> ← 0;
      PSL<CUR_MOD> ← saved_PSL<CUR_MOD>;
      PSL<PRV_MOD> ← saved_PSL<PRV_MOD>;
      PSL<IS> ← saved_PSL<IS>;
      PSL<IPL> ← saved_PSL<IPL>;
      PSL<N,Z,V,C> ← saved_PSL<N,Z,V,C>;
      end;
   otherwise                  ! (Other exceptions.)
      begin
      PSL<CM,TP,FPD,DV,FU,IV,T,N,Z,V,C> ← 0;
      PSL<CUR_MOD> ← 0;
      PSL<PRV_MOD> ← saved_PSL<CUR_MOD>;
      if vector<1:0> EQLU 1
         then PSL<IS> ← 1
         else PSL<IS> ← saved_PSL<IS>;
      if vector<1:0> EQLU 1
         then PSL<IPL> ← 31
         else PSL<IPL> ← saved_PSL<IPL>;
      end;
   end;

! If necessary, save the current stack pointer and load a
! new one.
if saved_PSL<IS> EQLU 0 then
   begin
   IPR[saved_PSL<CUR_MOD>] ← SP;
   SP ← IPR[ PSL<IS>'PSL<CUR_MOD> ];
   end;

! Push PC, the saved PSL, and any parameters onto the new
! stack, in the new mode.  If a memory management fault
! occurs while pushing onto the new stack, initiate an
! abort or halt.  (This will not occur with CHMx or
! instruction emulation exceptions, since they explicitly
! probe the destination stack and fault if it is invalid
! or inaccessible.)
-(SP) ← saved_PSL;
-(SP) ← PC;
{push parameters if any};

! Load PC with the address of the exception or interrupt
! handler.
PC ← vector<31:2> ' 0<1:0>;
{enable interrupts};
! Software interrupt clear the software-interrupt-pending
! bit.
if {software interrupt} then SISR< PSL<IPL> > ← 0;
```

Condition codes

```
N ← 0;
Z ← 0;
V ← 0;
C ← 0;
```

Exceptions

```
kernel-stack not valid
interrupt-stack not valid
```

Description

The vector associated with the exception or interrupt is read from the system control block. The current PSL is saved and a new PSL is created and loaded. If this is an interrupt, the new PSL has all fields cleared except IS and IPL. IPL is raised to the priority level of the interrupt request. IS is set to 1 if the low bits of the vector contain 01 (binary); otherwise, it is unchanged from the old PSL. If this is a CHMx exception, CUR_MOD is loaded with the new mode, PRV_MOD is loaded with the old value of CUR_MOD, IS and IPL are retained from the old PSL, and all other fields are cleared.

If this is an emulation exception, CUR_MOD, PRV_MOD, IS, IPL, and the condition codes are all retained from the old PSL, and all other fields are cleared. If this is any other kind of exception, PRV_MOD is loaded with the old value of CUR_MOD, IS and IPL are loaded according to the low bits of the vector, and all other fields are cleared. If the low bits of the vector are 01 (binary), IS is loaded with 1 and IPL is raised to 31; otherwise, IS and IPL are retained from the old PSL. Unless the processor is already running on the interrupt stack, the old stack pointer is saved and a new one is loaded. The saved PSL and the PC are pushed on the stack, along with any exception parameters. PC is loaded with the address of the interrupt or exception service routine indicated by bits ⟨31:2⟩ of the vector. Table 5.7 summarizes the state transitions CHMx, REI, interrupts, and exceptions cause.

Notes

1. Interrupts are disabled during this sequence.
2. On a fault or interrupt, the saved condition codes are UNPREDICTABLE; they are saved only to the extent necessary to ensure correct completion of the instruction when resumed.
3. After an abort, all the explicit and implicit operands of the aborted instruction are UNPREDICTABLE. The PC pushed on the stack points to the opcode of the aborted instruction, unless the instruction modified PC in a way that produces UNPREDICTABLE results.
4. After an abort, fault, or interrupt that pushes a PSL with FPD

set, the general registers except PC, SP, and FP are UNPREDICTABLE unless the instruction description specifies a setting. If FP is the destination in this case, it is also UNPREDICTABLE. On a kernel-stack-not-valid abort, both SP and FP are UNPREDICTABLE. This implies that processes stopped with FPD set cannot be resumed on processors of a different type or engineering change level.

5. If the processor gets an access-control-violation or translation-not-valid condition while attempting to push information on the kernel stack, a kernel-stack-not-valid abort is initiated instead, and IPL is raised to 31. The PSL and PC saved on the interrupt stack are those that would have been pushed on the kernel stack by the original exception. Additional information, if any, associated with the original exception is lost. If vector⟨1:0⟩ for kernel-stack-not-valid abort is 0, the operation of the processor is UNDEFINED. (Kernel stack not valid will not occur with CHMx or instruction emulation exceptions, since they explicitly probe the destination stack and fault if it is invalid or inaccessible.)
6. If the processor gets an access-control-violation or translation-not-valid condition while attempting to push information on the interrupt stack, the processor is halted and only the state of ISP, PC, and PSL is ensured to be correct for subsequent analysis. The PSL and PC have the values that would have been pushed on the interrupt stack.
7. The value of PSL⟨TP⟩ that is saved on the stack depends on the cause of the interrupt or exception and is as follows:

fault	clear
trace	clear
interrupt	clear (if FPD set) from PSL⟨TP⟩ (if after traps, before trace)
abort	UNPREDICTABLE
trap	from PSL⟨TP⟩
CHMx	from PSL⟨TP⟩
BPT, XFC	clear
reserved or privileged instruction	clear

8. The value of PC that is saved on the stack depends on the cause of the interrupt or exception and points to the following:

fault	instruction faulting
trace	next instruction to execute (instruction at the beginning of which the trace fault was taken)
interrupt	instruction interrupted or next instruction to execute
abort	instruction aborting or detecting kernel-stack-not-valid (not ensured on machine check)
trap	next instruction to execute
CHMx	next instruction to execute
BPT, XFC	BPT, XFC instruction
reserved or privileged instruction	privileged instruction

Table 5.7 summarizes the state transitions caused by interrupts, exceptions, and REI.

Table 5.7
Processor State Transitions

	Final State					
Initial State	User Mode	Super Mode	Exec Mode	Kernel Stack, IPL = 0	Kernel Stack, IPL > 0	Interrupt Stack
User Mode	REI or CHMU	CHMS	CHME	CHMK or e(0)[a]	i(0)[b]	e(1)[c] or i(1)[d]
Super Mode	REI	REI, CHMU, or CHMS	CHME	CHMK or e(0)	i(0)	e(1) or i(1)
Exec Mode	REI	REI	REI, CHMU, CHMS, or CHME	CHMK or e(0)	i(0)	e(1) or i(1)
Kernel Stack, IPL = 0	REI	REI	REI	REI, CHMx, LDPCTX, e(0), or MTPR IPL	MTPR IPL or i(0)	e(1), i(1), or SVPCTX
Kernel Stack, IPL > 0	REI	REI	REI	REI or MTPR IPL	REI, CHMx, LDPCTX, e(0), or i(0)	e(1), i(1), or SVPCTX
Interrupt Stack	REI	REI	REI	REI	REI or LDPCTX	REI, SVPCTX, MTPR IPL, exception, or interrupt

[a] e(0) indicates an exception with vector(1:0) = 0
[b] e(1) indicates an exception with vector(1:0) = 1
[c] i(0) indicates an interrupt with vector(1:0) = 0
[d] i(1) indicates an interrupt with vector(1:0) = 1

5.12 *Instructions Related to Exceptions and Interrupts*

This section describes instructions that are integral to the exception and interrupt architecture of the VAX processor. Software uses Return from Exception or Interrupt (REI) to return from an exception or interrupt service routine and resume normal execution. The Change Mode (CHMx) instructions allow software to request a change in the current access mode by initiating an exception that invokes a privileged handler.

REI Return from Exception or Interrupt

Format

```
opcode {(SP)+.r*}
```

Operation

```
tmp1 ← (SP)+;          ! Pick up saved PC
tmp2 ← (SP)+;          ! and PSL

! Check that tmp2 is a valid PSL
if {tmp2<IS> EQLU 1 AND tmp2<IPL> EQLU 0} OR
   {tmp2<IPL> GTRU 0 AND tmp2<CUR_MOD> NEQU 0} OR
   {tmp2<PRV_MOD> LSSU tmp2<CUR_MOD>} OR
   {tmp2<PSL_MBZ> NEQU 0} then {reserved operand fault};

if {compatibility mode implemented} then
   begin
   if {tmp2<CM> EQLU 1} AND
      {{tmp2<FPD,IS,DV,FU,IV> NEQU 0} OR
      {tmp2<CUR_MOD> NEQU 3}} then
      {reserved operand fault};
   end
else if {tmp2<CM> EQLU 1} then {reserved operand fault};

if {tmp2<CUR_MOD> LSSU PSL<CUR_MOD>} OR
   {tmp2<IS> EQLU 1 AND PSL<IS> EQLU 0} OR
   {tmp2<IPL> GTRU PSL<IPL>} then
      {reserved operand fault}

if PSL<IS> EQLU 1 then ISP ← SP  !save old stack pointer
         else IPR[ PSL<CUR_MOD> ] ← SP;
if PSL<TP> EQLU 1 then tmp2<TP> ← 1;        !TP ← TP or
               ! stack TP
PC ← tmp1;
PSL ← tmp2;
if PSL<IS> EQLU 0 then
   begin
   SP ← IPR[ PSL<CUR_MOD> ];        !switch stack
```

```
    if PSL<CUR_MOD> GEQU ASTLVL          !check for AST
                  ! delivery
       then {request interrupt at IPL 2};
    end;
{check for software interrupts};
{clear instruction look-ahead};
```

Condition codes

```
N ← saved PSL<3>;
Z ← saved PSL<2>;
V ← saved PSL<1>;
C ← saved PSL<0>;
```

Exceptions

reserved operand

Opcodes

02 REI Return from Exception or Interrupt

Description

A longword is popped from the current stack and held in a temporary PC. A second longword is popped from the current stack and held in a temporary PSL. The popped PSL is checked for internal consistency. The popped PSL is compared with the current PSL to make sure that the transition from current PSL to popped PSL is allowed. The current stack pointer is saved and a new stack pointer is selected according to the new PSL ⟨CUR_MOD⟩ and ⟨IS⟩ fields (see section 5.10.3). The level of the highest-privileged AST is checked against the current mode to see whether a pending AST can be delivered (see Chapter 6). Execution resumes with the instruction being executed at the time of the exception or interrupt.

After completing an REI, a processor will correctly execute a modified instruction stream (see section 7.2.2).

Notes

1. The exception or interrupt service routine is responsible for restoring any registers saved and removing any parameters from the stack.
2. As usual for faults, if translation not valid or access-control violation occurs while popping PC or PSL from the stack, the stack pointer is restored as part of the initiation of the fault.
3. REI to compatibility mode results in a reserved operand fault if compatibility mode is not implemented.

CHMx Change Mode

Format

```
opcode code.rw {-(ySP).w*}
                where y = MINU(x, PSL<CUR_MOD>)
                where x = Mode selected by opcode
                          (K=0, E=1, S=2, U=3)
```

Operation

```
tmp1 ← {mode selected by opcode (K=0, E=1, S=2, U=3)};
tmp2 ← MINU(tmp1, PSL<CUR_MOD>);   !maximize privilege
tmp3 ← SEXT(code);
if {PSL<IS> EQLU 1} then HALT;     !illegal from I stack

IPR[ PSL<CUR_MOD> ] ← SP;          !save old stack pointer
tmp4 ← IPR[ tmp2 ];                !get new stack pointer
! Check new stack access
PROBEW (from tmp4-1 through tmp4-12 with mode=tmp2);
if {access-control violation} then
   {initiate access-control-violation fault};
if {translation not valid} then
   {initiate translation-not-valid fault};
if {modify} then
   {initiate modify fault};

{initiate CHMx exception with new_mode=tmp2
   and parameter=tmp3
   using 40+tmp1*4 (hex) as SCB offset
   using tmp4 as the new SP
   and not storing SP again};
```

Condition codes

```
N ← 0;
Z ← 0;
V ← 0;
C ← 0;
```

Exceptions

halt

Opcodes

BC	CHMK	Change Mode to Kernel
BD	CHME	Change Mode to Executive
BE	CHMS	Change Mode to Supervisor
BF	CHMU	Change Mode to User

Description

Change-mode instructions allow processes to change their access mode in a controlled manner. The instruction only increases privilege (decreases the access mode) or leaves it unchanged.

A change in mode also results in a change of stack pointers: the old pointer is saved, the new pointer is loaded. The PSL, PC, and code passed by the instruction are pushed on the stack of

the new mode. The saved PC addresses the instruction following the CHMx instruction. The code is sign extended. Figure 5.5 illustrates the new stack's appearance after execution.

The destination mode selected by the opcode is used to obtain a location from the system control block. This location addresses the CHMx dispatcher for the specified mode. If the vector⟨1:0⟩ code is NEQU 0, then the operation is UNDEFINED.

Notes

1. As usual for faults, any memory management fault saves PC and PSL and leaves SP as it was at the beginning of the instruction (except for any pushes onto the kernel stack). Note, though, that addresses just off the top of the target stack may have been written (see section 5.10.5).
2. By software convention, negative codes are reserved to DIGITAL and DIGITAL's customers.

Examples

```
CHMK    #7          ; Request the kernel-mode service
                    ; specified by code 7.

CHME    #4          ; Request the executive-mode service
                    ; specified by code 4.

CHMS    #-2         ; Request the supervisor-mode service
                    ; specified by customer code -2.
```

6

Process Structure

A process is a single thread of execution. It is the basic scheduling entity that is executed by the processor. A process consists of an address space and both hardware and software context. The hardware context of a process is defined by a process control block (PCB) that contains images of the 14 general-purpose registers; the processor status longword (PSL); the program counter (PC); the four per-process stack pointers (USP, ESP, KSP, SSP); the base and length registers P0BR, P0LR, P1BR, and P1LR used to access the process virtual memory provided to the process; and several minor control fields. For a process to execute, the majority of the PCB must be moved into the internal registers. While a process is executing, some of its hardware context is being updated in the internal registers. When a process is not being executed, its hardware context is stored in a data structure termed the "process control block." Saving the contents of the privileged registers in the PCB of the currently executing process and then loading a new context from another PCB is termed "context switching." Context switching occurs as one process after another is scheduled for execution.

6.1 Process Context

Shown in Figure 6.1 is the process control block for the currently executing process. The PCB is pointed to by the process control block base (PCBB) register, an internal privileged register. Figure 6.2 shows the PCBB. When the processor is initialized, the contents of PCBB are UNPREDICTABLE.

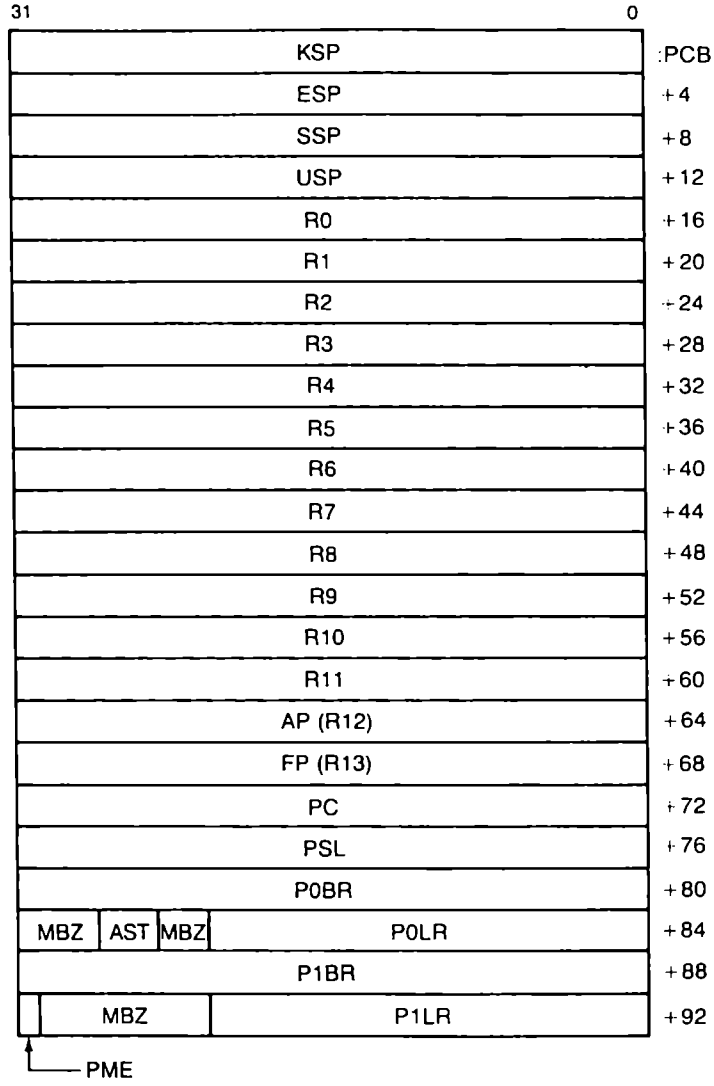


Figure 6.1
Process Control Block (PCB)

For a processor with a 24-bit, 30-bit, 32-bit one-to-one, or 32-bit sign-extended physical address mode, the processor locates the PCB in physical memory and holds a physical address in PCBB. For a processor with a 34-bit physical address mode, the processor locates the PCB in virtual memory and holds a virtual address in PCBB. If the processor does hold a virtual address in PCBB, the following must be true or the operation of the processor is

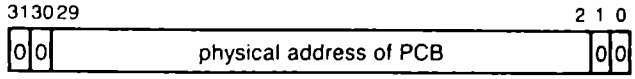


Figure 6.2
Process Control-Block Base Register (PCBB)

UNDEFINED: the PCB must not cross a page boundary; the page containing the PCB must be valid; the page must allow kernel write access; and (on processors that implement modify fault) the page must be marked "modified."

Software must ensure that there is not more than one PCB in any aligned 128-byte block of memory (physical or virtual memory, depending on whether PCBB contains a physical or a virtual address).

The PCB contains all of the switchable process context collected into a compact form for ease of movement to and from the privileged internal registers. Although in any normal operating system there is additional software context for each process, the following description is limited to that portion of the PCB known to the hardware. The PCB's contents are described in Table 6.1. To alter its P0BR, P1BR, P0LR, P1LR, ASTLVL, or PME, a process must be executing in kernel mode. The process must first store the desired new value in the memory image of the PCB, then move the value to the appropriate privileged register. This protocol results from the fact that these are read-only fields (for the context switch instructions) in the PCB.

The ASTLVL and PME fields of the PCB may be contained in internal processor registers when the process is running.

6.1.1 *Performance Monitor Enable Register*

The performance-monitor-enable (PME) register controls a signal visible to an external hardware performance monitor. PME allows the system to identify those processes for which monitoring is desired and so permits their behavior to be observed without interference caused by the activity of other processes. Figure 6.3 shows PME. Processor initialization sets PME to zero.

Table 6.1
Contents of the Process Control Block

Name	Mnemonic	Offset (hex)	Extent
kernel stack pointer	KSP	0	⟨31:0⟩
executive stack pointer	ESP	4	⟨31:0⟩
supervisor stack pointer	SSP	8	⟨31:0⟩
user stack pointer	USP	C	⟨31:0⟩
general registers	R0–R13	10–44	⟨31:0⟩
program counter	PC	48	⟨31:0⟩
processor status longword	PSL	4C	⟨31:0⟩
P0 base register	P0BR	50	⟨31:0⟩
P0 length register	P0LR	54	⟨21:0⟩
AST level	ASTLVL	54	⟨26:24⟩
P1 base register	P1BR	58	⟨31:0⟩
P1 length register	P1LR	5C	⟨21:0⟩
performance monitor enable	PME	5C	⟨31⟩

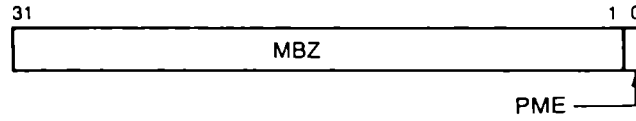


Figure 6.3
Performance-Monitor-Enable Register (PME)

6.2 *Asynchronous System Traps*

Asynchronous system traps (AST) are a technique for notifying a process of events that are not synchronized with its execution and for initiating processing of asynchronous events with the least possible delay. This delay in delivery of the AST may be due to either process non-residence or an access mode mismatch. The efficient handling of ASTs in the VAX system requires some hardware assistance to detect changes in access mode (PSL⟨CUR_MOD⟩). A process in any of the four access modes (kernel, executive, supervisor, and user) may receive ASTs; however, an AST for a less privileged access mode must not be permitted to interrupt execution in a more protected access mode. Since outward access mode transitions occur only in the REI instruction, comparison of the current access mode field is made with a privileged register,

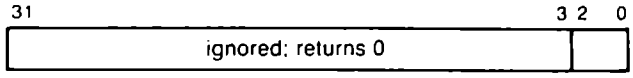


Figure 6.4
AST-Level Register (ASTLVL)

ASTLVL, shown in Figure 6.4. ASTLVL contains the most privileged access mode number for which an AST is pending. If the new access mode is greater than or equal to the pending ASTLVL, an IPL 2 interrupt is posted to cause delivery of the pending AST.

The software flow for AST processing follows:

1. An event associated with an AST causes software enqueuing of an AST control block to the software PCB, and the software sets the ASTLVL field in the hardware PCB to the most privileged access mode for which an AST is pending. If the target process is currently executing, the ASTLVL privileged register also has to be set.
2. When an REI instruction detects a transition to an access mode that can be interrupted by a pending AST, an IPL 2 interrupt is triggered to cause delivery of the AST. Note that the REI instruction does not make pending AST checks while returning to a routine executing on the interrupt stack.
3. The (IPL 2) interrupt service routine should compute the correct new value for ASTLVL that prevents additional AST delivery interrupts while in kernel mode and move that value to the PCB and the ASTLVL register before lowering IPL and actually dispatching the AST. This interrupt service routine normally executes on the kernel stack in the context of the process receiving the AST.
4. At the conclusion of processing for an AST, the ASTLVL is again computed and moved to the PCB and ASTLVL register by software.

If ASTLVL contains 4, no AST is pending for the current process. If ASTLVL is less than 4, an AST is pending for the mode corresponding to the value of ASTLVL. Values of ASTLVL greater than 4 are reserved. Execution of *MTPR src, #PR$_ASTLVL* with src⟨31:0⟩ GEQU 5 results in UNDEFINED behavior. The preferred implementation is to cause reserved-operand fault. Processor initialization sets ASTLVL to 4. Note that loading ASTLVL with

MTPR does not affect SISR or request a software interrupt. Those effects of ASTLVL occur only during REI.

6.3 Process-Scheduling Interrupts

Two of the software interrupt priorities are reserved for process scheduling software:

IPL 2 (AST delivery interrupt) is triggered by an REI that detects PSL<CUR_MOD> GEQU ASTLVL and indicates that a pending AST may now be delivered for the currently executing process.

IPL 3 (process-scheduling interrupt) is triggered by software. It indicates that a process has changed software priority and that the process scheduler should reschedule to find the highest priority executable process to run.

6.4 Process-Structure Instructions

Process scheduling software must execute on the interrupt stack (PSL<IS> set) to have a non-context-switched stack available for use. If the scheduler were running on a process's kernel stack, any state information it had there would disappear when a new process is selected. The scheduler usually runs on the interrupt stack as the result of interrupts, which are typically serviced on the interrupt stack. However, some synchronous scheduling requests such as a WAIT service may want to cause rescheduling without any interrupt occurrence. For this reason, the save-process-context (SVPCTX) instruction can be issued while on either the kernel or the interrupt stack, because the instruction forces a transition to execution on the interrupt stack.

All of the process structure instructions are privileged and require kernel mode.

LDPCTX Load Process Context

Format

```
opcode {PCB.r*, -(KSP).w*}
```

Operation

```
if PSL<CUR_MOD> NEQU 0
   then {privileged instruction fault};
if PSL<IS> NEQU 1 then {UNDEFINED};

{invalidate per-process translation buffer
   entries on scalar processor};

! The PCB is located by the address in PCBB.
if {internal registers for stack pointers} then
   begin
   KSP ← (PCB);
   ESP ← (PCB+4);
   SSP ← (PCB+8);
   USP ← (PCB+12);
   end;
R0 ← (PCB+16);
R1 ← (PCB+20);
R2 ← (PCB+24);
R3 ← (PCB+28);
R4 ← (PCB+32);
R5 ← (PCB+36);
R6 ← (PCB+40);
R7 ← (PCB+44);
R8 ← (PCB+48);
R9 ← (PCB+52);
R10 ← (PCB+56);
R11 ← (PCB+60);
AP ← (PCB+64);
FP ← (PCB+68);

tmp1 ← (PCB+80);
if {tmp1<1:0> NEQU 0} then {UNDEFINED};
P0BR ← tmp1;
if (PCB+84)<31:27> NEQU 0 then {UNDEFINED};
if (PCB+84)<23:22> NEQU 0 then {UNDEFINED};
P0LR ← (PCB+84)<21:0>;
if (PCB+84)<26:24> GEQU 5 then {UNDEFINED};
ASTLVL ← (PCB+84)<26:24>;
tmp1 ← (PCB+88);
tmp2 ← tmp1 + 2**23;
if {tmp2<1:0> NEQU 0} then {UNDEFINED};
P1BR ← tmp1;
if (PCB+92)<30:22> NEQU 0 then {UNDEFINED};
P1LR ← (PCB+92)<21:0>;
```

LDPCTX

```
PME ← (PCB+92)<31>;

ISP ← SP;              ! Save the interrupt stack pointer.
{interrupts off};
PSL<IS> ← 0;           ! Change from the interrupt stack
SP ← (PCB);            ! to the new kernel stack.
{interrupts on};
-(SP) ← (PCB+76);      ! Push PSL onto kernel stack.
-(SP) ← (PCB+72);      ! Push PC onto kernel stack.
                       ! (If kernel stack is inaccessible
                       ! or invalid, then UNDEFINED.)
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

```
reserved operand
privileged instruction
```

Opcodes

06 LDPCTX Load Process Context

Description

If the processor is not in kernel mode, a privileged-instruction fault is taken. Otherwise, the processor loads the process state in the process control block specified by the privileged PCBB register. The general registers, process-space memory management registers, and address space number register (if implemented) are loaded from the PCB into the scalar processor. Execution switches to the kernel stack. The PC and PSL are moved from the PCB to the stack, suitable for use by a subsequent REI instruction. The process-space TB state is unconditionally flushed.

Notes

1. Some processors keep a copy of each of the per-process stack pointers in internal registers. In those processors, LDPCTX loads the internal registers from the PCB. Other processors do not keep a copy of all four per-process stack pointers in internal registers. Rather such processors keep only the stack pointer for the current access mode in an internal register and switch this with the PCB contents whenever the current access mode changes.
2. The preferred implementation of UNDEFINED operation is reserved operand abort.
3. LDPCTX does not invalidate per-process TB state in the TB of the vector processor. To invalidate TB state on the vector

processor, use the TBIA, TBIS, or VTBIA internal processor registers.

4. LDPCTX does not load the memory management registers of the vector processor, if such copies reside there.
5. To guarantee correct operation, a LDPCTX must be followed by an REI instruction.

SVPCTX Save Process Context

Format

```
opcode {(SP)+.r*, PCB.w*}
```

Operation

```
if PSL<CUR_MOD> NEQU 0 then
   {privileged instruction fault};
!PCB is located by address in PCBB
if {internal registers for stack pointers} then
   begin
   (PCB) ← KSP;
   (PCB+4) ← ESP;
   (PCB+8) ← SSP;
   (PCB+12) ← USP;
   end;
(PCB+16) ← R0;
(PCB+20) ← R1;
(PCB+24) ← R2;
(PCB+28) ← R3;
(PCB+32) ← R4;
(PCB+36) ← R5;
(PCB+40) ← R6;
(PCB+44) ← R7;
(PCB+48) ← R8;
(PCB+52) ← R9;
(PCB+56) ← R10;
(PCB+60) ← R11;
(PCB+64) ← AP;
(PCB+68) ← FP;
(PCB+72) ← (SP)+;                    !pop PC
(PCB+76) ← (SP)+;                    !pop PSL

if PSL<IS> EQLU 0 then
   begin
   PSL<IPL> ← MAXU(1, PSL<IPL>);
   (PCB) ← SP;                       !save KSP
   KSP ← SP;
   {interrupts off};
   PSL<IS> ← 1;
   SP ← ISP;
   {interrupts on};
   end;
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions `privileged instruction`

Opcodes 07 SVPCTX Save Process Context

Description The process control block is specified by the privileged PCBB register. The general registers are saved into the PCB. The PC and PSL currently on the top of the current stack are popped and stored in the PCB. If a SVPCTX instruction is executed when the processor is executing on the kernel stack, then the processor switches to the interrupt stack and IPL is maximized with 1.

Notes

1. The per-process memory management registers, ASTLVL, and PME from the PCB are not saved because they are rarely changed. Thus, not writing them saves overhead.
2. Some processors keep a copy of each of the per-process stack pointers in internal registers. In those processors, SVPCTX stores the internal registers into the PCB. Other processors do not keep a copy of all four per-process stack pointers in internal registers. Rather these processors keep only the stack pointer for the current access mode in an internal register and switch this with the PCB contents whenever the current access mode changes.
3. Between the SVPCTX instruction that saves state for one process and the LDPCTX that loads the state of another, the internal stack pointers may not be referenced by MFPR or MTPR instructions. This implies that interrupt service routines invoked at a priority higher than the lowest one used for context switching must not reference the process stack pointers.

Example

The following example illustrates how the process structure instructions can be used to implement process dispatching software. It is assumed that this simple dispatch routine is always entered with an interrupt.

```
; ENTERED WITH INTERRUPT
; IPL=3

RESCHED: SVPCTX                    ; Save context in PCB
         .
         .
         .
         <Set state to runnable>
         <and place current PCB>
         <on proper RUN queue.>
```

SVPCTX

```
.
.
.
<Remove head of highest>
<priority, non-empty,>
<RUN queue.>

MTPR @#PCB, #PR$_PCBB    ; Set PCB address in PCBB
LDPCTX                   ; Load context from PCB
                         ; for new process
REI                      ; Place process in execution
```

7

Memory and I/O

The layout and detailed structure of physical memory and I/O space is implementation dependent, but the overall structure of VAX memory and I/O is part of the architecture.

7.1 Memory, Multiprocessing, and Interprocessor Communication

A VAX system comprises one or more VAX processors and one or more I/O devices that all share a common memory. The VAX architecture specifies how these elements communicate with each other. It specifies how data is shared between the processors and how the processors signal each other. It includes rules for the ordering of read and write operations and interrupts and for mutual exclusion to avoid data corruption by simultaneous reads and writes from multiple processors.

There are three interrelated topics:

- Signaling and data visibility. Each processor must signal the next processor or I/O device when the next activity should begin. Each processor must also ensure that the next processor or I/O device will be able to read the new values of any shared data when the signal is received. There are three principal means for signaling and ensuring data visibility: interlocked instructions, requesting an interrupt, and writing to I/O space.
- Read and write ordering. In a uniprocessor, a read returns the value of the most recent write to the same location. In a VAX multiprocessor system, no time standard is held in

common across the several processors, so terms based on time, such as "most recent," have little utility. VAX systems use the more basic concept of the ordering of events, particularly the ordering of read and write accesses and interrupts, to coordinate their activities.

- Mutual exclusion. Multiple processes can compete for access to shared data. Unless something prevents it, more than one process may simultaneously access a shared data resource in a "critical section." Their mutual actions can interfere with each other, producing incoherent results, that is, the required relations among the data no longer hold. Mutual exclusion protocols resolve access conflicts to critical sections so that the accessed data remain coherent.

  VAX hardware provides primitives for the software to implement various mutual exclusion protocols. Those hardware primitives are atomic read operations and atomic write operations for some kinds of memory references, and the "interlocked" instructions for atomic read-modify-write memory references. Software can use the hardware primitives to construct many synchronization elements such as barriers, semaphores, mutual-exclusion locks, and token passing. The interlocked instructions directly implement mutual exclusion to critical sections.

The interlocked instructions have functions involving data visibility, signaling, read and write ordering, and mutual exclusion. They are central to communication and control in a VAX multiprocessor.

A vector processor synchronizes only with its paired scalar processor, as described in section 13.3.2. To synchronize two vector processors with each other, the first vector processor synchronizes with its paired scalar processor, the two scalar processors synchronize with each other according to the rules of this chapter, and finally the second scalar processor synchronizes with its paired vector processor.

The term "I/O device" includes any I/O processor, device, controller, or adapter that has control status registers in I/O space or that can read or write the common memory.

7.1.1 *Interprocessor Signaling and Data Visibility*

The signals listed below ensure that all observers can read the signal itself and any previous writes made by the signaling processor or I/O device. The maximum delay from when a program issues a signal until the signal takes effect in the other processors is machine dependent. If a program or I/O device does not issue one of the signals, it is UNPREDICTABLE whether another observer can read the new values of all the previously written shared data. The signals are as follows:

1. Any interlocked instruction with its modify operand in memory space or any I/O device's interlocked read-modify-write sequence to memory space.
2. Any interrupt request from a processor to another processor or from an I/O device to a processor.
3. Any processor (but not I/O device) write access to I/O space, including a read-modify-write access but excluding any vector-write access. This ensures that I/O devices can observe any processor data written previously to I/O space. It also ensures that a processor can't write memory, start I/O, have the I/O complete and signal a second processor, and have the second processor unable to observe the writes in memory.

   There are two functions involving the console that are also interprocessor signals.
4. Processor entry to console mode as the result of a HALT (from kernel mode or from the front console panel) or an error halt.
5. Any console function that changes memory state. (Most VAX consoles do not signal that a change has been made. The changes themselves, however, are made visible.)

As an example of making data visible using an interlocked instruction, consider two processors, P1 and P2. Variable X is initialized to 0, and P2 waits for P1 to change it to 1.

```
        P1                              P2
        --                              --
        MOVB    #1,X            P$2:    CMPB    #1,X
        ADAWI   #0,Z                    BNEQ    P$2
P$1:    BRB     P$1                     HALT
```

The interlocked instruction (ADAWI) makes the new value of X visible to processor P2. When P2 observes the value of 1 take effect in location X, P2 halts. If the ADAWI instruction in P1

were not there, the new value of X might never become visible to P2, and P2 could loop forever and never halt.

7.1.2 Atomicity and Corruption of Shared Data

A common memory subsystem allows data to be shared among the processors and I/O devices of a VAX system. Multiple simultaneous accesses may cause some data to become "corrupted" while other "atomic" accesses keep the data intact. The following two definitions precisely describe atomicity and corruption:

- *Atomicity.* The access of a shared location in memory by one processor or I/O device is classified as atomic (indivisible) or non-atomic (divisible) with respect to an access of the same location by another processor or I/O device. A memory access is atomic when any other memory access to the same location may occur before or after but not during the first access. A memory access is non-atomic when another memory access to the same location may occur before, during, or after the first access.
- *Corruption.* A shared location accessed from memory non-atomically can become corrupted, that is, the location when written can become any byte-wise combination of all the data that is concurrently being written to it, or when a shared location is read, the value read may be any byte-wise combination of its original value with any new values that are concurrently being written to it. Corruption is also known as "data incoherency" and "word tearing."

  Corruption can occur two ways: (1) from the non-atomic accesses of two or more writers, in which case the corrupted value will remain in the memory location until some program reinitializes it; and (2) from the non-atomic accesses of a single writer and one or more readers, in which case only the read accesses are corrupted, and subsequent read accesses will return an uncorrupted value. A location (or its value) cannot become corrupted if it is not shared, if it is always read and written by atomic accesses, or if it is accessed by multiple readers and no writers.

To support shared data, a VAX system meets the following three requirements:

1. The memory ensures that the granularity of access for independent modification by any processor or I/O device is the

byte. This does not imply a maximum reference size of one byte but only that independent modifying accesses to adjacent bytes produce the same results regardless of the order of execution. For example, suppose locations 1000 and 1001 contain the values 5 and 6, respectively. Suppose one processor executes INCB 1000 and another executes INCB 1001. Then, regardless of the order of execution, including effectively simultaneous execution, the final contents must be 6 and 7.

2. When executing an instruction, a VAX processor always reads and always writes certain operands from memory by atomic accesses. These operands, termed "atomic operands," are the following: a byte operand, an aligned word operand, an aligned longword operand (including vector longword elements), a bit-field contained within one byte, and an aligned longword address pointed to by a displacement deferred mode or autoincrement deferred mode operand specifier. An atomic operand is the only kind of shared data for which a VAX processor necessarily provides atomic memory access.

   A modify operand is both read and written by the same instruction. The read and the write are separate accesses to the same address. Unless the modify operand is part of an interlocked instruction (see the next section), another processor or I/O device can access the operand between the read access and the write access, even if both are atomic.

3. A processor is not required to implement any of the following as atomic operands: any operand that is not naturally aligned, quadwords (including D_floating and G_floating data and vector elements, and queue-entry headers and links), octawords (including H_floating data), character strings, numeric strings, bit-fields not contained within one byte, packed decimal strings, or vectors. Thus, for most VAX implementations, these operands will be read or written from memory by non-atomic accesses.

7.1.3 *Using Interlocks to Prevent the Corruption of Shared Data*

VAX processors do not provide atomic memory access to all data. To prevent the corruption of data accessed non-atomically, readers and writers must access them in a mutually exclusive manner – at any one time, there must be either only one writer to shared data or any number of readers. There must be neither multiple writers

nor both a reader and a writer. Failure to use mutual exclusion when multiple processors or I/O devices are reading or writing with non-atomic accesses produces UNPREDICTABLE results.

Seven interlocked instructions provide the hardware primitives for software to implement various mutual-exclusion protocols. The interlocked instructions test and set a memory-resident control variable, such as a lock or a semaphore, through hardware-implemented atomic memory accesses: BBSSI, BBCCI, ADAWI, INSQHI, INSQTI, REMQHI, and REMQTI. A control variable is a bit string in shared memory whose value indicates whether access to the associated shared data is allowed. Software, by using interlocked instructions, sets a control variable to any of the various values that indicate whether the shared data is accessible or inaccessible to other processors and I/O devices. BBSSI and BBCCI test and set a 1-bit control variable. ADAWI maintains a count in an aligned-word control variable. The INSQHI, INSQTI, REMQHI, and REMQTI instructions use an interlock on the queue header to allow queues to be maintained consistently. The control variable for these queue instructions resides in bit ⟨0⟩ of the queue header.

Each of these instructions accesses a control variable using an atomic sequence of operations termed an "interlocked sequence." A processor wishing to begin this sequence requests access to the interlock grain, which is an implementation-dependent region of physical memory containing the control variable. When the access has been granted, the interlocked sequence starts and the interlock grain is locked (made inaccessible) to the interlocked sequences of all other processors and I/O devices. When the sequence is completed, the interlock grain is unlocked, making it accessible to the interlocked sequences of other processors and I/O devices. Thus, any two interlocked sequences performed on the same interlock grain are always atomic with respect to each other, that is, they will always interlock. Note that interlocking occurs at the level of physical memory.

The following rules apply to interlocks:

1. Non-interlocked instructions are not necessarily blocked from reading or writing a locked memory region; whether they are blocked is implementation dependent.

2. The aligned longword is the basic unit of interlocking. Memory hardware ensures that interlocked sequences interlock when the starting byte addresses of their control variables map to the same aligned longword in physical memory. It is UNPREDICTABLE whether interlocked sequences will interlock if they reference control variables whose lowest-order bytes are not in the same aligned longword. The memory hardware determines the size of an interlock grain. The size may vary from a longword to all of memory and may even be discontiguous: for example, every 2^{16}th longword may be locked.

   For example, suppose a programmer chooses to use a word in the following way: byte 1000 will be a byte counter, and byte 1001 will contain a 1-bit lock. Then, even though the control variable for an ADAWI is an entire word, the programmer must not expect that a BBSSI to byte 1001 will interlock with an ADAWI to the word, since only byte 1000 need be locked for the ADAWI and byte 1001 for the BBSSI.
3. It is possible in VAX implementations that interlocked sequences on two different control variables will unintentionally interlock because both control variables lie within the same interlock grain. For example, if the size of the interlock grain for a VAX processor is one page (512 bytes), then an interlocked sequence performed on a control variable within a page will lock out all other interlocked sequences that are performed on the same page.
4. A processor may read more bytes than necessary when reading a control variable for an interlocked sequence, but it only writes back the following: for BBSSI and BBCCI, the byte that contains the control variable; for ADAWI, the aligned word that contains the control variable; and for the interlocked queue instructions, the aligned longword that contains the control variable.
5. If software accesses a control variable with an interlocked instruction, then software must write the control variable only with interlocked instructions. If a non-interlocked write occurs while an interlocked sequence is in progress on the control variable, the data may be overwritten when the interlocked control variable is rewritten. Thus, when a control variable is tested or modified, it is UNPREDICTABLE whether a non-interlocked instruction updates any byte contained within a range of bytes that is written.

For example, suppose the following: the initial contents of a word-sized control variable, CTRL, are ^X0000. Processor 1 executing *ADAWI #1, CTRL* begins an interlocked sequence on CTRL as processor 2 executing *MOVB #8, CTRL+1* starts to write to CTRL+1. Then, when both instructions complete, the final value of CTRL could be either ^X0001 or ^X0801.

There are two good programming practices that result from the above requirement:

- If you use interlocked instructions to set or clear flag bits in a byte, use interlocked instructions for all writes to that byte.
- If you use an interlocked instruction to allocate a resource, use an interlocked instruction to deallocate it.

6. No interlocked sequence is ever performed on a GPR. Therefore, BBSSI, BBCCI, and ADAWI control variables that need to be interlocked must not be in GPRs.
7. ADAWI is the only interlocked instruction whose control variable can be located in I/O space, in which case it must also be in Unibus space. When it is in I/O space but not Unibus space, it is implementation dependent whether an interlocked sequence is ever performed on the control variable. See section 7.5.3.
8. For performance reasons, software should avoid looping with an interlocked instruction when waiting for access to shared data. Recall that non-interlocked instructions are not necessarily blocked from reading control variables in a locked interlock grain. On most VAX processors, once an interlocked instruction determines that shared data is not available, repeated checking for availability is most efficiently done with non-interlocked instructions. If the non-interlocked instruction indicates the data is available, then an interlocked instruction can confirm the result; otherwise, the program can continue looping on the non-interlocked instruction or do something else. Note that another processor can gain control of the data in the time between the non-interlocked and interlocked checks for data availability. For further details and an example, see the instruction descriptions of BBSSI and BBCCI in Chapter 3.

7.2 *Instruction Stream Processing and Modification*

This section describes the processing of instruction streams in the VAX architecture and how to ensure correct processing behavior when the instruction stream is modified.

The VAX architecture assumes a sequential control flow model of instruction stream processing. In this model, instructions are processed from the instruction stream, which resides in instruction memory, one at a time; no instruction is processed (even partially) until the one before it completes.

An instruction consists of its opcode, all operand specifiers, and any operands that are necessarily contiguous with the rest of the instruction stream. Thus, the instruction stream includes short literals, immediate-mode operands, absolute addresses in absolute mode addressing, branch displacements, CALLx entry masks, and CASEx tables. Except for immediate-mode operands and branch displacements, the instruction stream does not include operands, even ones addressed relative to the PC. Except for absolute addresses, the instruction stream does not include indirect addresses. It also does not include EDITPC patterns. All instruction operands and indirect addresses not considered to be within the instruction stream are considered data.

Instruction memory is those portions of memory used for the storage and retrieval of instructions to be executed by the processor.

No portion of instruction memory may reside in page tables, PCBs, or the SCB. Instructions also may not lie within the following region above the top-of-stack because the region is allowed to change in an unpredictable manner: the region starts at –1(SP) and ends at either –5*512(SP) or the first non-writable page (within the current mode) above the top-of-stack, whichever comes first. The operation of the processor is UNPREDICTABLE when it executes instructions retrieved from the above mentioned regions.

The processing of one instruction entails the following actions: translating the instruction address using the appropriate page table entry; fetching the instruction from instruction memory with the translated address; decoding the instruction and fetching required operands; executing the instruction; and storing the re-

sults, if any, in the indicated locations. Thus, the processing of an instruction stream requires various resources such as instruction memory, page tables, general purpose registers, and privileged registers.

7.2.1 *Prefetching of Instructions*

Since a strict hardware implementation of the sequential control flow model yields poor performance, many VAX processors overlap the processing of instructions to increase their speed. This performance improvement is made possible by utilizing instruction lookahead and buffering logic, translation buffers, and multiple processors. Instruction lookahead and buffering logic includes, but is not limited to, instruction buffers, virtual instruction caches, and physical instruction and data caches. Note that in utilizing these techniques, VAX processors are still generally required to maintain to software the sequential control flow model of instruction stream processing. However, VAX processors are allowed to deviate from this model during instruction stream modification (discussed in the next section) when changes to instruction memory do not necessarily update a processor's prefetched copy of the same.

7.2.2 *Instruction Stream Modification*

An instruction stream is modified whenever its contents are changed, added to, or deleted. This modification can occur when changing any of the resources used in determining the location, content, or mapping of the instruction stream (listed in Table 7.1). For example, writing data into instruction memory, changing the value of a valid PTE, or changing memory management mode modifies the instruction stream.

However, changing any of the above resources does not always modify the instruction stream. For example, decrementing P0LR by 4 to exclude an invalid PTE at the end of the P0 page table does not modify the instruction stream. The determination of which changes to the above resources actually cause instruction stream modification is often complex; therefore, it is recommended that software treat any change to these resources as causing an instruction stream modification, if there is any doubt.

For VAX processors that implement lookahead and buffering logic, modification of the instruction stream can cause the processor's

Table 7.1
Resources That When Changed Can Modify the Instruction Stream

Resource	How Changed	Software Action Necessary After the Change When Instruction-Stream Modification Occurs
Instruction Memory	Writes[a]	Issue REI[b].
Valid Process PTEs	Writes[a]	If MME=1 then software must flush stale process PTEs from the TB and issue REI[b].
PxBR, PxLR	MTPR	If MME=1 then software must flush stale process PTEs from the TB and issue REI[b].
PxBR, PxLR	LDPCTX	Issue REI[b].
Valid System PTEs	Writes[a]	If MME=1 then software must flush both stale system and process PTEs from the TB and issue REI[b].
SBR, SLR	MTPR	If MME=1 then software must flush both stale system and process PTEs from the TB and issue REI[b].
MAPEN	MTPR	Issue REI[b]. Before enabling MAPEN, software must flush the entire TB. (Note that an REI should always be issued after changing memory management mode.)

[a]Writes to instruction operands (explicit or implicit) in memory or memory writes by I/O or another processor.
[b]Software must issue an REI after the instruction stream has been modified and before the modified instruction stream is processed (as seen by software under a sequential control flow model) using the changed instruction processing resource.

buffered copy of the instruction stream to become "stale," that is, no longer match the modified instruction stream in instruction memory. From the processor's point of view, the modification has created two equally valid instruction streams (the original one and the modified one) from which the processor can choose to fetch instructions. As a result, the processor may execute parts of both the unmodified and the modified instruction streams. This situation can go on indefinitely unless the processor is forced by software to stop using stale copies of the instruction stream.

Software can force the processor to stop using stale copies of the instruction stream and begin to fetch instructions only from the modified instruction stream by using the REI instruction. To ensure correct results, whenever software modifies the instruction stream by changing any of the resources listed in Table 7.1 on a given processor, software must issue an REI on that processor after the change and before the processor actually uses the changed resource (as seen by software under a sequential control flow model)

to process the modified instruction stream. Furthermore, software must flush stale PTEs from that processor's translation buffer before issuing the REI if the changed resource pertains to memory management.

Until software issues the REI and flushes stale PTEs from the translation buffer (when appropriate), it is UNPREDICTABLE whether the modified or unmodified instruction stream or bytes of each will be executed by the affected processor. Note that if software changes the next instruction location to be executed in the instruction stream, it is always UNPREDICTABLE whether the old or the new instruction in that location will be executed.

If the changed resource is used immediately after the instruction stream modification for the processing of subsequent instructions, then software must ensure that these instructions and the required REI reside in the same locations in both the unmodified and the modified instruction streams. For example, if software changes the mapping for the current page of the instruction stream by writing into the page's PTE, then software can force the processor to use the updated instruction memory by guaranteeing that the next two instructions after the PTE write are an MTPR-to-TBIS (to flush the old PTE value out of the TB) followed by REI. It does this by placing an MTPR-to-TBIS and an REI in the next locations to be executed by the processor under either the old or the new PTE mapping.

In the case of a multiprocessor configuration, software must apply the above rule to each processor in turn. Software must flush stale PTEs from the TB and issue an REI on each processor whose instruction stream has been modified.

Note that flushing PTEs from the TB by writing to TBIA or TBIS does not modify the instruction stream, since the PTEs in the TB are merely copies of the PTEs in memory. Thus, instructions fetched using a particular TB entry are still valid after the TB entry is flushed as long as the underlying PTE in the page table is unmodified. If this PTE in memory is modified, then the process state is changed and software must issue an REI instruction as described earlier in this chapter and in Table 7.1.

Executing an REI on most VAX processors is a time-consuming operation. To minimize execution time, software should avoid issuing excessive REIs. When multiple instruction stream modifications are necessary, software should perform them all first and then issue a single REI afterward. For example, if a debugger is inserting BPT instructions into the instruction stream, it should do all necessary changes and then issue an REI when finished.

Lastly, the above rules do not apply to a compatibility mode procedure, which can modify its instruction stream without any additional synchronization.

7.3 *Memory References*

The memory references made by each instruction (and therefore the possible memory exceptions) are specified as part of the VAX architecture. Any required or permitted memory reference (read, modify, or write) may be made more than once, except for references to I/O space, which are made once and only once. Operands requiring interlocked access are always referenced. In general, for operands not requiring interlocked access, it is UNPREDICTABLE whether an operand is referenced if it does not affect the result (including condition codes). Further clarifications and exceptions to this simplified rule are listed below. Software must not rely on the occurrence of memory management exceptions on operands that do not affect the result of an instruction. The probe instructions should be used to determine the accessibility of a memory location. In general, no results are written unless the instruction can be completed or can be suspended with PSL⟨FPD⟩ set (but see section 5.10.5).

1. It is UNPREDICTABLE whether longwords containing indirect addresses are read. For example, *MULL3 #0, @16(R5), A* may or may not access the longword containing the address of the second operand.
2. If a branch is not taken, it is UNPREDICTABLE whether the branch displacement is read.
3. It is UNPREDICTABLE whether all bytes for .r operands are read. For example, TSTF may read only the word containing the sign and exponent. BLBC and BLBS may read only the low byte of the source operand.
4. All bytes for .w operands are always written.

5. It is UNPREDICTABLE whether all bytes for .m operands are either read (with modify intent) or written. However, a modify operand requiring interlocked read and write is always accessed. For example, *ADDL2 #0, A* may read only A (without modify intent). *INCL A* may write only the bytes of A that changed. The sum operand of *ADAWI #0, A* is always read and written back interlocked.
6. For .a operands (and for .v operands when .v is not a register), the memory reference behavior is peculiar to each instruction or instruction group. Overriding the rules given below, it is UNPREDICTABLE whether an otherwise unreadable operand is read if it appears as an immediate mode operand. For example, *PUSHAB (R0)* cannot read the byte at (R0), but *PUSHAB #512* can read the value 512.

   a. POLY{F,D,G,H}. If the argument is not zero, each entry in the coefficient table is read unless an arithmetic exception occurs before the instruction completes. If the argument is zero, it is UNPREDICTABLE whether the entire table or only the last coefficient is read.
   b. MOVA{B,W,L,Q,O} and PUSHA{B,W,L,Q,O}. The address operand is not referenced.
   c. Field Instructions (EXTV, EXTZV, INSV, CMPV, CMPZV, FFS, FFC). The aligned longword(s) containing the field specified by FIELD (pos, size, base) can be read. For INSV, only this aligned longword(s) can be written. It is UNPREDICTABLE whether all or some of the bytes in these longwords are accessed. Furthermore, no memory references occur for a zero-length bit string.
   d. BB{S,C}, BB{S,C}{S,C}. Only the single byte containing the test bit specified by the base and position operands is read. If the test bit does not need to change state, it is UNPREDICTABLE whether the byte is written back.
   e. BB{SS,CC}I. Only the single byte containing the test bit specified by the base and position operands is referenced using the interlocked forms of read and write. The test bit is written even if its state is unchanged.
   f. JMP and JSB. The address is not referenced by the JMP or JSB (but will be read as instruction stream data for the next instruction).
   g. CALL{S,G}. The two bytes (containing the entry mask) at

the destination address are read. The argument list for CALLG is not referenced.

h. Interlocked Queue. It is UNPREDICTABLE whether the backward pointer of the queue header is accessed for INSQHI, REMQHI.

i. Character-String instructions (CMPC, LOCC, MATCHC, MOVC, MOVTC, MOVTUC, SCANC, SKPC, SPANC). No memory references occur for a zero-length character string.

j. Decimal instructions (ADDP, ASHP, CMPP, CVTLP, CVTPL, CVTPS, CVTPT, CVTSP, CVTTP, DIVP, MOVP, MULP, SUBP). No memory references occur for a zero-length trailing numeric string.

7. Some of the character-string instructions (MOVTUC, CMPC3, CMPC5, SCANC, SPANC, LOCC, SKPC, and MATCHC) can stop before the whole source string is processed. Three definitions help define the required memory references for these instructions.

   The stop byte is the byte that ends the instruction execution without using the string length end condition. It is the last byte on which the answer of the instruction depends. (The stop byte may have any position in the string, including first or last, or it may not exist at all. For string matches, it is the last byte of the matched string.)

   A source string consists of a body concatenated with a tail.

   The body of a source string is the substring from the first byte up to and including the stop byte, if one exists, or up to and including the last byte (as determined by the source string's length) if no stop byte exists. (The body may be null only if the source string has a zero length.)

   The tail of a source string is the substring from the first byte after the body up to and including the last byte in the source string as determined by the source string's length. (The tail will be null if there is no stop byte or if the stop byte is the last byte.)

   Character strings are defined by length and starting address. Some strings (ASCIZ strings) are delimited by a specific character. The "real" length of the string is not known, and 64K is used as the length. Only some of the VAX character-string instructions can be reasonably used on character delimited

strings. These instructions are MOVTUC, SPANC, SCANC, LOCC, and SKPC. For these five instructions, it is necessary to guarantee that no memory management exceptions will occur beyond the page containing the delimiting character. The absence of such a requirement could cause a program that works on one processor to fail on another because of access violations on data that is not necessary to produce the correct result.

For string operands specified by length and starting address, one of the following rules applies:

a. For MOVC3, MOVTC, and CRC, all bytes are referenced. These instructions have no end condition other than string length.
b. For MOVC5, the stop byte is defined as the last byte moved from the source string. MOVC5 references all bytes except when the source string is longer than the destination string; in the latter case, no bytes in the source string's tail beyond the page containing the stop byte are referenced.
c. For CMPC3, CMPC5, and MATCHC, all bytes in a string's body are referenced. It is UNPREDICTABLE whether any bytes in a string's tail are referenced.
d. For MOVTUC, SCANC, SPANC, LOCC, and SKPC, all bytes in the source string's body are referenced, and no bytes in the source string's tail beyond the page containing the stop byte are referenced. For MOVTUC, the destination address that would receive the translated escape character is not written into, nor is any larger address written into.

For table operands, one of the following rules applies:

a. In the table for MOVTC, MOVTUC, SCANC, and SPANC, entries are accessed for the corresponding source characters or values. It is UNPREDICTABLE whether the other table entries are accessed.
b. For the CRC table operand, it is UNPREDICTABLE whether all or only part of the table is accessed.

8. If a packed decimal source string contains invalid digits, it is UNPREDICTABLE whether the entire source string is read and whether any or all of the destination is written.

   If there are no invalid digits in a packed decimal source string, one of the following rules applies:

a. EDITPC, MOVP, ADDP6, SUBP6, MULP, DIVP, CVTPT, CVTTP, CVTPS, CVTSP, and ASHP. All bytes of the source strings are read, and all bytes of the result are written, unless an exception condition is detected and the instruction can be completed without reading all the bytes in the source strings.
b. CMPP3 and CMPP4. It is UNPREDICTABLE whether all bytes of the two source strings are read.
c. ADDP4 and SUBP4. All bytes of the addend (or subtrahend) string are read. It is UNPREDICTABLE whether all bytes of the result are written.
d. CVTLP. All bytes of the destination string are written.
e. CVTPL. All bytes of the source string are read.
f. EDITPC, CVTPT, CVTTP. The table entries are accessed for the corresponding source bytes. It is UNPREDICTABLE whether the other table entries are accessed.

9. PROBER and PROBEW. The first and last bytes specified by the base and length operand are not accessed.

7.4 *Cache*

A hardware implementation may include a mechanism to reduce access time by making local copies of recently used memory contents. Such a mechanism is termed a "cache." A cache must be implemented in such a way that its existence is transparent to software (except for timing and error reporting, control, and recovery). In particular, the following must be true:

1. An I/O transfer from memory to a peripheral, started after a program write to the same memory, must output the updated memory value.
2. A program memory read, executed after the completion of an I/O transfer from a peripheral to the same memory, must read the updated memory value.
3. If one processor writes or modifies memory and then executes HALTs, a read or modify of the same memory by another processor must read the updated value.
4. If a processor writes or modifies memory and then halts as a result of power failure, a read or modify of the same memory must read the updated value (provided that the duration of the power failure does not exceed the maximum non-volatile period of the main memory).

5. In multiprocessor systems, access to variables shared between processors must be interlocked by software executing one of the interlocked instructions (BBSSI, BBCCI, ADAWI, INSQHI, INSQTI, REMQHI, REMQTI).
6. Valid accesses to I/O registers must not be cached.
7. A cache may prefetch instructions or data. In a virtual cache, memory management exception conditions could occur during prefetch. Such exceptions should not be taken until the prefetched data is referenced by an instruction.

Processor initialization must leave the cache either empty or valid.

7.5 *I/O Structure*

The VAX I/O architecture is very similar to the PDP-11 structure. The principal difference is the method by which internal processor registers (such as the memory management registers) are accessed. Peripheral device control and status registers and data registers appear at locations in part of the physical address space (I/O space), and can therefore be manipulated by most memory reference instructions. Use of general instructions permits all the virtual address mapping and protection mechanisms described in Chapter 4 to be used when referencing I/O registers. Note: Implementations that include a cache feature must suppress caching for references in the I/O space. The physical address space consists of two parts: memory space and I/O space. Memory space starts at address zero and continues to an implementation-dependent limit. I/O space begins at that limit and continues to the end of the physical address space.

For any member of the VAX series implementing the UNIBUS interconnect, there will be one or more areas of I/O space, each 2^{18} bytes in length, that "map through" to UNIBUS addresses. The collection of these areas is referred to as the UNIBUS space.

7.5.1 *Use of PTEs by I/O Devices*

Some I/O devices, such as those using the VAX CI Port Architecture, use VAX page tables to translate addresses. These I/O devices use the PTE formats described in Chapter 4, including particular field encodings of invalid PTEs. Specifically, I/O devices for VAX processors with 34-bit physical addresses decode PTE⟨31,26,25⟩

Table 7.2
PTE Types for Use by I/O

PTE Type	PTE⟨31,26,25,22⟩
Processor-valid PFN	1 x x x
Resident-for-I/O PFN	0 0 0 0[a]
Global Page Table Index	0 0 1 1[a]
I/O Abort	0 1 x x

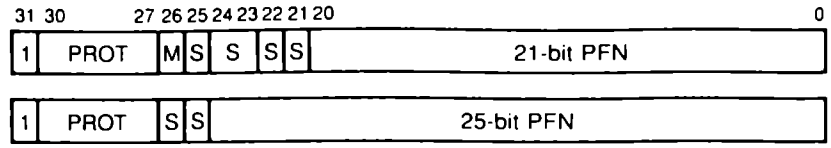
[a]PTE⟨22⟩ is a copy of PTE⟨25⟩ for older 30-bit I/O devices using 21-bit PFNs.

and use a 25-bit page frame number (PFN); I/O devices for VAX processors with 30-bit physical addresses decode PTE⟨31,26,25⟩ (or PTE⟨31,26,22⟩ for older devices) and use a 21-bit PFN. The PTE bits are decoded into four combinations, as shown in Table 7.2. Some of these are used in the same way as in the VAX-processor PTE formats, and some are used in different ways.

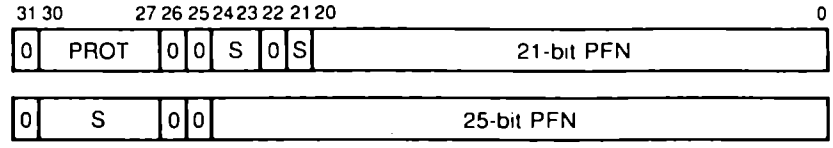
When PTE⟨31⟩ = 1, on systems with 34-bit physical addresses PTE⟨24:0⟩ is a 25-bit PFN, and on systems with 30-bit physical addresses PTE⟨20:0⟩ is a 21-bit PFN. These two formats are shown in Figure 7.1. PTE⟨PROT⟩ and the PFN fields are identical to those used by VAX processor hardware, and they are used the same way. The mapped pages may be used by either the processor or I/O devices.

When PTE⟨31,26,25⟩ = 000 (binary), on systems with 34-bit physical addresses PTE⟨24:0⟩ is a 25-bit PFN, and on systems with 30-bit physical addresses PTE⟨20:0⟩ is a 21-bit PFN. On 30-bit systems, PTE⟨22⟩ is a copy of PTE⟨25⟩ so that older 30-bit I/O devices can decode PTE⟨31,26,22⟩ = 000 (binary) to get the 21-bit PFN. These formats are shown in Figure 7.1. The formats are not used by VAX processor hardware. I/O devices can use the formats only if they have established a cooperating protocol with the operating system. Then the operating system will guarantee that the mapped pages remain memory resident as long as the I/O is still operating.

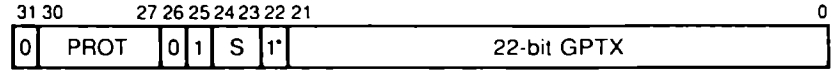
When PTE⟨31,26,25⟩ = 001 (binary), PTE⟨21:0⟩ is a 22-bit global page table index (GPTX). On 30-bit systems, PTE⟨22⟩ is a copy of PTE⟨25⟩ so that older 30-bit I/O devices can decode PTE⟨31,26,22⟩



PTEs with Processor-Valid Page Frame Number (PTE⟨31⟩ = 1)



PTEs with Resident-for-I/O Page Frame Number (PTE⟨31,26,25⟩ = 000)



* PTE⟨22⟩ is a 1 for devices that decode PTE⟨31,26,22⟩, that is, older devices using 21-bit PFNs. Otherwise it is "reserved for software."

PTEs with Global Page Table Index (PTE⟨31,26,25⟩ = 001)

I/O Abort (PTE⟨31,26⟩ = 01)

Figure 7.1
PTEs for Use by I/O Devices

= 001 (binary) to get the 22-bit GPTX. This format is shown in Figure 7.1. The format is not used by VAX processor hardware. I/O devices can use the format only if they have established a co-operating protocol with the operating system. The I/O device has a global-page-table-base register (GBR) that is loaded (via memory) by software with a system virtual address. The I/O device calculates GBR + (GPTX × 4) to get the system virtual address of a second PTE. The second PTE must have a valid PFN and must have its decoded bits equal to either 1xx or 000 (binary). If either of these requirements is not met, the result is UNDEFINED.

When PTE⟨31,26⟩ = 01 (binary), as shown in Figure 7.1, the PTE format is reserved to DIGITAL. I/O devices will abort in a device-dependent manner.

I/O devices that decode PTE⟨31,26,25⟩, either 34-bit or 30-bit devices, never look at or check PTE⟨PROT⟩ or modify PTE⟨M⟩. I/O devices that decode PTE⟨31,26,22⟩, that is, older 30-bit devices, may look at and check PTE⟨PROT⟩, depending on the device, but they never modify PTE⟨M⟩.

The I/O devices described in this section do memory mapping using the same system page table as the VAX processor, but they have their own copies of the processor's SBR and SLR, which the software loads by writing to memory.

I/O-device buffer addresses are described in terms of a system virtual address of the PTE for the first buffer page and a byte offset within that page. In addition, the I/O devices use a global page table in memory and an I/O hardware global-page-table-base register, which the software loads by writing to memory.

7.5.2 *Restrictions on I/O Registers*

The following is a list of both hardware and programming constraints on I/O registers. These items affect both hardware register design and programming considerations.

1. The physical address of an I/O register must be an integral multiple of the register size in bytes (which must be a power of two); that is, all registers must be aligned to natural address boundaries.
2. References using a length attribute other than the length of the register or references to unaligned addresses may produce UNPREDICTABLE results. For example, a byte reference to a word-length register will not necessarily respond by supplying or modifying the byte addressed.
3. In all peripheral devices, error and status bits that may be asynchronously set by the device must be cleared by software writing a 1 to that bit position and are not affected by writing a 0. This is to prevent clearing bits that may be asynchronously set between reading and writing a register.
4. The following applies only to systems that support a UNIBUS. Only byte and word references of read-modify-write type (.mb or .mw access type) in UNIBUS I/O spaces are guaranteed to interlock correctly. Processors must ensure that .mb and .mw access types use the DATIP - DATO(B) functions when the operand physical address selects a UNIBUS device. This con-

straint does not apply to longword, quadword, field, all floating, or string operations if implemented using byte- or word-modifying references. This constraint also does not apply to instructions precluded from I/O space references. References in the I/O space other than in UNIBUS spaces are UNDEFINED with respect to interlocking. This includes the BBSSI and BBCCI instructions.

5. Any string, quadword, octaword, F_floating, D_floating, G_floating, H_floating, and field references in the I/O space result in UNDEFINED behavior.
6. Page tables must not be located in I/O space. References to page table entries located in I/O space result in UNDEFINED behavior.
7. The PCB and SCB must not be located in I/O space. References to the PCB or to SCB entries located in I/O space result in UNDEFINED behavior.
8. Unless otherwise stated as part of the device specification, a read from a device register after a write will return any values updated by the write.
9. Software must not depend on a write to I/O space completing with the instruction doing the write: a processor may continue instruction execution without waiting for the written data to reach its destination. The mechanism for ensuring the currency of the effects of a write is a read from the same device. (Note that I/O-space reads are much slower than writes since the processor must wait for round-trip signals from the I/O device.)
10. The time for an interrupt to occur after a program initiates device actions resulting in interrupts is UNPREDICTABLE. Also, the time for the last interrupt to occur after a device's interrupts have been disabled is UNPREDICTABLE. A program can set a software flag indicating that any interrupts for a device should be ignored.
11. The time for an I/O-system error to be reported is UNPREDICTABLE. Therefore, when the error report occurs, the PC can be far removed from the instruction that caused the error.
12. I/O devices that reference memory are always started by a write to I/O space and never by a read. This ensures that any data written previously (by the processor starting the I/O device) may be read by the device. (See section 7.1.1.)

7.5.3 *Instructions Usable to Reference I/O Space*

Some instructions must not be used for referencing I/O space. Reasons for this include:

1. String instructions are restartable with PSL(FPD).
2. The PC, SP, or PCBB cannot point to I/O space.
3. I/O space does not support operand types of quad, floating, field, or queue; nor can the position, size, length, or base of them be from I/O space.
4. The instruction may be interruptible because it is potentially a slow instruction in some implementations.
5. Only instructions with a maximum of one modify or write destination can be used. The destination must be the last operand.

Vector instructions are not allowed to reference I/O space. If a vector instruction references I/O space, the results are UNPREDICTABLE.

For any memory reference to I/O space, the programmer must use an instruction from the following lists and must ensure that no interrupts or exceptions will occur, including page fault, modify fault, or overflow trap, after the first I/O space reference. To ensure no interrupts, the programmer must avoid operand specifier modes 9, 11, 13, and 15, and these modes indexed. (Symbolically, these are @(Rn)+, @B^D(Rn), @W^D(Rn), and @L^D(Rn), and these indexed.) The hardware may allow interrupts for these modes to minimize interrupt latency. For the instructions in the following lists, the hardware ensures that no other interrupts will occur after the first I/O space access.

Since these instructions are not interruptible after I/O space accesses (except for the addressing modes above), their execution will extend the interrupt latency. The programmer should make some effort to keep them short by minimizing the number of memory references. For example, use R0 through R13 as instruction operands instead.

Instructions for which any explicit operand can be in I/O space are as follows:

```
ADAWI                  CHM{K,E,S,U}                  MTPR
ADD{B,W,L}2            CMP{B,W,L}                    PROBE{R,W}
ADD{B,W,L}3            CVT{BW,BL,WB,WL,LB,LW}        PUSHA{B,W,L}
ADWC                   DEC{B,W,L}                    PUSHAQ
BIC{B,W,L}2            INC{B,W,L}                    PUSHL
BIC{B,W,L}3            MCOM{B,W,L}                   SBWC
BICPSW                 MFPR                          SUB{B,W,L}2
BIS{B,W,L}2            MNEG{B,W,L}                   SUB{B,W,L}3
BIS{B,W,L}3            MOV{B,W,L}                    TST{B,W,L}
BISPSW                 MOVA{B,W,L}                   XOR{B,W,L}2
BIT{B,W,L}             MOVAQ                         XOR{B,W,L}3
CASE{B,W,L}            MOVPSL
CLR{B,W,L}             MOVZ{BW,BL,WL}
```

Instructions for which some operand can be in I/O space are as follows:

```
BLB{S,C}      (any operands but branch displacement)
XFC           (depending on implementation)
REMQUE addr   (destination)
REMQHI addr   (destination)
REMQTI addr   (destination)
```

Notwithstanding the above rules, it is possible for a specific hardware implementation to execute macro code from the I/O space or to allow the stack or PCB to be in I/O space. This might be used, for example, as part of the bootstrap process. If this is done, then it is valid for software to transfer to this code.

8

Privileged Registers

The internal processor register (IPR) space provides access to many types of CPU control and status registers such as the memory-management base registers, parts of the PSL, and the stack pointers. These registers are explicitly accessible only by the move-to-processor-register (MTPR) and move-from-processor-register (MFPR) instructions, which require kernel-mode access privileges.

Table 8.1 summarizes architecturally required IPRs. Those internal processor registers that require further explanation are described below. Reference to general registers means R0 through R13, the SP, and the PC (see Chapter 1). Registers referenced by the MTPR and MFPR instructions are designated processor registers and appear in the processor register space.

8.1 Per-Process Registers and Context Switching

Several per-process registers are loaded from the PCB during a context load operation and, with the exception of the memory management registers, address space number register, PME, and AST level, are written back to the PCB during a context save operation (see Chapter 6). Some implementations may copy some or all of these registers from the PCB into scratchpad registers and write them back into the PCB during a context save operation. Other implementations may retain the registers in main memory in the PCB.

```
ADAWI                CHM{K,E,S,U}                MTPR
ADD{B,W,L}2          CMP{B,W,L}                  PROBE{R,W}
ADD{B,W,L}3          CVT{BW,BL,WB,WL,LB,LW}      PUSHA{B,W,L}
ADWC                 DEC{B,W,L}                  PUSHAQ
BIC{B,W,L}2          INC{B,W,L}                  PUSHL
BIC{B,W,L}3          MCOM{B,W,L}                 SBWC
BICPSW               MFPR                        SUB{B,W,L}2
BIS{B,W,L}2          MNEG{B,W,L}                 SUB{B,W,L}3
BIS{B,W,L}3          MOV{B,W,L}                  TST{B,W,L}
BISPSW               MOVA{B,W,L}                 XOR{B,W,L}2
BIT{B,W,L}           MOVAQ                       XOR{B,W,L}3
CASE{B,W,L}          MOVPSL
CLR{B,W,L}           MOVZ{BW,BL,WL}
```

Instructions for which some operand can be in I/O space are as follows:

```
BLB{S,C}      (any operands but branch displacement)
XFC           (depending on implementation)
REMQUE addr   (destination)
REMQHI addr   (destination)
REMQTI addr   (destination)
```

Notwithstanding the above rules, it is possible for a specific hardware implementation to execute macro code from the I/O space or to allow the stack or PCB to be in I/O space. This might be used, for example, as part of the bootstrap process. If this is done, then it is valid for software to transfer to this code.

8

Privileged Registers

The internal processor register (IPR) space provides access to many types of CPU control and status registers such as the memory-management base registers, parts of the PSL, and the stack pointers. These registers are explicitly accessible only by the move-to-processor-register (MTPR) and move-from-processor-register (MFPR) instructions, which require kernel-mode access privileges.

Table 8.1 summarizes architecturally required IPRs. Those internal processor registers that require further explanation are described below. Reference to general registers means R0 through R13, the SP, and the PC (see Chapter 1). Registers referenced by the MTPR and MFPR instructions are designated processor registers and appear in the processor register space.

8.1 Per-Process Registers and Context Switching

Several per-process registers are loaded from the PCB during a context load operation and, with the exception of the memory management registers, address space number register, PME, and AST level, are written back to the PCB during a context save operation (see Chapter 6). Some implementations may copy some or all of these registers from the PCB into scratchpad registers and write them back into the PCB during a context save operation. Other implementations may retain the registers in main memory in the PCB.

Table 8.1
Architected Internal Processor Registers (Part 1)

Name	Mnemonic	Decimal	Hex	Type[a]	One Copy Per[b]
kernel stack pointer	KSP	0	0	R/W	process
executive stack pointer	ESP	1	1	R/W	process
supervisor stack pointer	SSP	2	2	R/W	process
user stack pointer	USP	3	3	R/W	process
interrupt stack pointer	ISP	4	4	R/W	CPU
P0 base register[c]	P0BR	8	8	R/W	process [VP]
P0 length register[c]	P0LR	9	9	R/W	process [VP]
P1 base register[c]	P1BR	10	A	R/W	process [VP]
P1 length register[c]	P1LR	11	B	R/W	process [VP]
system base register[c]	SBR	12	C	R/W	CPU [VP]
system length register[c]	SLR	13	D	R/W	CPU [VP]
CPU identification[d]	CPUID	14	E	R	CPU
process control block base	PCBB	16	10	R/W	CPU

[a]An "R" indicates that a register can be read but not written; "W" that it can be written but not read; "R/W" that it can be both read and written.
[b]The word "process" indicates one copy per process, residing on the scalar processor, loaded by LDPCTX; "process [VP]" one copy per process on the scalar processor and optionally one more copy on the vector processor, loaded into the scalar processor by LDPCTX; "CPU" one copy per scalar processor, not affected by LDPCTX; "CPU/VP" one copy per scalar/vector pair, not affected by LDPCTX; "VP" one copy per vector processor, not affected by LDPCTX; "CPU [VP]" one copy per scalar processor and optionally one more copy on the vector processor, not affected by LDPCTX.
[c]This register is not required on a vector processor. For more detail, refer to Chapter 12.
[d]This register is optional. For more detail, refer to Chapter 11.

An implementation may retain some or all per-process stack pointers only in the PCB. In this case, MTPR and MFPR for these registers must access the corresponding PCB locations. However, implementations that have per-process stack pointers in hardware scratchpads are not required to access the corresponding PCB locations for MTPR and MFPR. The PCB locations get updated when a SVPCTX instruction is executed.

Implementations are not allowed to retain copies of the per-process memory-management registers (P0BR, P0LR, P1BR, P1LR) only in the PCB. The processor must implement each of these registers in hardware.

Table 8.1
Architected Internal Processor Registers (Part 2)

Name	Mnemonic	Decimal	Hex	Type[a]	One Copy Per[b]
system control block base	SCBB	17	11	R/W	CPU
interrupt priority level	IPL	18	12	R/W	CPU
AST level	ASTLVL	19	13	R/W	process
software interrupt request	SIRR	20	14	W	CPU
software interrupt summary	SISR	21	15	R/W	CPU
interval clock control[c]	ICCS	24	18	R/W	CPU
next interval count[c]	NICR	25	19	W	CPU
interval count[c]	ICR	26	1A	R	CPU
time of year[c]	TODR	27	1B	R/W	CPU
console receiver status[c]	RXCS	32	20	R/W	CPU
console receiver data buffer[c]	RXDB	33	21	R	CPU
console transmit status[c]	TXCS	34	22	R/W	CPU
console transmit data buffer[c]	TXDB	35	23	W	CPU

[a]An "R" indicates that a register can be read but not written; "W" that it can be written but not read; "R/W" that it can be both read and written.
[b]The word "process" indicates one copy per process, residing on the scalar processor, loaded by LDPCTX; "process [VP]" one copy per process on the scalar processor and optionally one more copy on the vector processor, loaded into the scalar processor by LDPCTX; "CPU" one copy per scalar processor, not affected by LDPCTX; "CPU/VP" one copy per scalar/vector pair, not affected by LDPCTX; "VP" one copy per vector processor, not affected by LDPCTX; "CPU [VP]" one copy per scalar processor and optionally one more copy on the vector processor, not affected by LDPCTX.
[c]This register is optional (however, a subset of ICCS is required). For more detail, refer to Chapter 11.

It is possible that some implementations will retain ASTLVL, PME, or both only in the PCB. These processors will implement MTPR and MFPR for those registers as a no-op, at least in the sense that the destination operand (for MFPR) or IPR (for MTPR) is not written. Other implementations may copy either or both of these registers from the PCB into scratchpad registers.

The SVPCTX instruction does not write the per-process memory-management registers, ASTLVL, and PME back into the PCB. To ensure that the PCB is always correctly updated, software must use the following convention when referencing any of these registers.

Table 8.1
Architected Internal Processor Registers (Part 3)

Name	Mnemonic	Decimal	Hex	Type[a]	One Copy Per[b]
memory management enable	MAPEN	56	38	R/W	CPU
translation buffer invalidate all	TBIA	57	39	W	CPU/VP
translation buffer invalidate single	TBIS	58	3A	W	CPU/VP
performance monitor enable[c]	PME	61	3D	R/W	process
system identification	SID	62	3E	R	CPU
translation buffer check	TBCHK	63	3F	W	CPU
vector processor status register[d]	VPSR	144	90	R/W	VP
vector arithmetic exception register[d]	VAER	145	91	R	VP
vector memory activity check[d]	VMAC	146	92	R	VP
vector TB invalidate all[d]	VTBIA	147	93	W	VP
vector state address register[d]	VSAR	148	94	R/W	VP

[a]An "R" indicates that a register can be read but not written; "W" that it can be written but not read; "R/W" that it can be both read and written.
[b]The word "process" indicates one copy per process, residing on the scalar processor, loaded by LDPCTX; "process [VP]" one copy per process on the scalar processor and optionally one more copy on the vector processor, loaded into the scalar processor by LDPCTX; "CPU" one copy per scalar processor, not affected by LDPCTX; "CPU/VP" one copy per scalar/vector pair, not affected by LDPCTX; "VP" one copy per vector processor, not affected by LDPCTX; "CPU [VP]" one copy per scalar processor and optionally one more copy on the vector processor, not affected by LDPCTX.
[c]This register is optional. For more detail, refer to Chapter 11.
[d]This register is not required in implementations that do not support a vector processor. For more detail, refer to Chapter 12.

1. *Write.* Software must first write the value directly into the proper location in the current PCB by using a MOVL, for example, then execute an MTPR with the same source as the MOVL. Implementations that do not retain internal copies of these registers will effectively no-op the MTPR instruction. They will not take a reserved operand fault, which would normally occur for a non-existent register.
2. *Read.* Software can read the value directly from the proper location in the current PCB by using a EXTZV, for example.

It is not necessary to execute an MFPR from the corresponding internal register, since the PCB location always contains an updated value due to the software convention for writing these registers.

8.2 *Stack-Pointer Images*

Reference to the stack pointer general register (SP) will access one of five possible stack pointers—user, supervisor, executive, kernel, or interrupt—depending on the values of PSL⟨CUR_MOD⟩ and PSL⟨IS⟩ (see Chapter 5). Additionally, software can access any of the five stack pointers (including the one currently selected by PSL⟨CUR_MOD⟩ and PSL⟨IS⟩) with the MTPR and MFPR instructions (even on processors that implement the KSP, SSP, ESP, or USP only in the PCB). Results are correct even if the stack pointer specified by the current mode and IS bits in the PSL is referenced in the internal processor register address space by an MTPR or MFPR instruction. This means that an MFPR or MTPR to the KSP (if PSL⟨IS⟩=0) or the ISP (if PSL⟨IS⟩=1) is equivalent to a MOVL from or to the SP.

8.3 *MTPR and MFPR Instructions*

The following instructions allow privileged software to access all internal processor registers and thereby monitor and control the operation of the VAX processor.

MTPR Move To Processor Register

Format

```
opcode        src.rl, procreg.rl
```

Operation

```
if PSL <CUR_MOD> NEQ 0 then {privileged-instruction fault};
IPR[procreg] ← src;
```

Condition codes

```
N ← UNPREDICTABLE
Z ← UNPREDICTABLE
V ← UNPREDICTABLE
C ← UNPREDICTABLE
```

Exceptions

```
privileged instruction
```

Opcodes

DA MTPR Move To Processor Register

Description

MTPR loads the source operand specified by source into all copies of the processor register specified by procreg that are implemented on the scalar and vector processors. The procreg operand is a longword that contains the processor register number. Execution may have register-dependent side effects.

Notes

1. A privileged instruction fault occurs if instruction execution is attempted in other than kernel mode (PSL⟨CUR_MOD⟩ EQLU 0).
2. If a register is implemented only as a PCB location, MTPR to that register has no effect.
3. In kernel mode, the operation of the processor is UNDEFINED after execution of MTPR to a read-only register, MTPR to a non-existent register, MTPR of a non-zero value to an MBZ field, or MTPR of a reserved value to a register. The preferred implementation is to cause a reserved-operand fault.
4. After an MTPR instruction, the condition codes are UNPREDICTABLE, unless noted otherwise under the description of the specific processor register.

MFPR Move From Processor Register

Format

```
opcode procreg.rl, dst.wl
```

Operation

```
if PSL <CUR_MOD> NEQ 0 then {privileged-instruction
   fault};
dst ← IPR[procreg];
```

Condition codes

```
N ← UNPREDICTABLE
Z ← UNPREDICTABLE
V ← UNPREDICTABLE
C ← UNPREDICTABLE
```

Exceptions

```
privileged instruction
```

Opcodes

DB MFPR Move From Processor Register

Description

The destination operand is replaced by the contents of the processor register specified by procreg. The procreg operand is a longword that contains the processor register number. Execution may have register-dependent side effects.

Notes

1. A privileged instruction fault occurs if instruction execution is attempted in other than kernel mode. (PSL⟨CUR_MOD⟩ EQLU 0).
2. If a register is implemented only as a PCB location, MFPR from that register has no effect.
3. In kernel mode, the operation of the processor is UNDEFINED after execution of MFPR from a register that does not exist or after execution of MFPR from a write-only register. The preferred implementation is to cause a reserved-operand fault.
4. After an MFPR instruction, the condition codes are UNPREDICTABLE, unless noted otherwise under the description of the specific processor register.

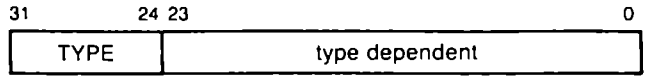


Figure 8.1
System Identification Register (SID)

8.3.1 *CPU Identification Register*

In a multiprocessor system, software occasionally must know on what processor it is running. The CPU identification register (CPUID) exists for this purpose. CPUID contains a different value in each processor of a multiprocessor system and is in the range 0..K–1 in a K-processor system.

8.3.2 *System Identification Register*

The system identification register (SID) specifies the processor type and includes an implementation-dependent field. The processor type field is used by software when handling features that vary among processor implementations. The implementation-dependent field typically specifies additional information, such as hardware revision level and microcode revision level, and is included in the error log to more finely distinguish processor types. The SID is shown in Figure 8.1. Table 8.2 shows the processor type codes.

8.3.3 *System-Type Register*

The system-type register (SYS_TYPE) augments the SID, which identifies processors. SYS_TYPE allows software to distinguish between different systems based on the same processor. It is implemented on systems that provide console functions by means of VAX instructions in I/O space. Systems that include separate console processors do not include SYS_TYPE, but may provide information for the same purpose through console functions. The SYS_TYPE⟨31:24⟩ subfield specifies the system type and distinguishes between different systems based on the same processor, all of which would have the same SID. Table 8.2 shows the system type codes assigned for each SID value. Figure 8.2 shows the SYS_TYPE register.

8.3.4 *Time-of-Year Clock Register*

The time-of-year clock is used to measure the duration of power failures and is required for unattended restart after a power failure.

Table 8.2
SID and SYS_TYPE Assignments (Part 1)

SID	Processor	SYS_TYPE	System
0	Reserved to DIGITAL		
1	VAX-11/780; VAX-11/782; VAX-11/785		
2	VAX-11/750		
3	VAX-11/730		
4	VAX 8600; VAX 8650		
5	VAX 8200, 8300; VAX 8250, 8350; VS8000		
6	VAX 8700, 8800, 8810, 8820-N; VAX 8500, 8530, 8550		
7	MicroVAX-I		
8	MicroVAX-II chip		
		0	Reserved to DIGITAL
		1	MicroVAX II
		2–3	Reserved to DIGITAL
		4	MicroVAX 2000; VAXstation 2000; PC/LAN SERVER 2000; MicroVAX Information Processor Module (aka Industrial (VAXserver, MicroVAX) 2000)
		5	VAXterm
		6	VAX 9000 Console
		7–127	Reserved to DIGITAL
		128–255	Reserved to system owners
9	Reserved to DIGITAL		
10	CVAX chip		
		0	Reserved to DIGITAL
		1	MicroVAX 3500 or 3600; VAXserver 3500 or 3600; VAXstation 3200 or 3500; MicroVAX 3300, 3400; VAXserver 3300, 3400; MicroVAX 3800, 3900; VAXserver 3800, 3900

The time-of-year clock consists of one longword register, shown in Figure 8.3. The register forms an unsigned 32-bit binary counter that is driven by a precision clock source with at least .0025 percent accuracy (approximately 65 seconds per month). The least significant bit of the counter represents a resolution of 10 milliseconds. Thus, the counter cycles to 0 after approximately 497 days.

Table 8.2
SID and SYS_TYPE Assignments (Part 2)

SID	Processor	SYS_TYPE	System
10	CVAX chip (continued)		
		2	VAX 62n0; VAX Fileserver 62n0; VAX 63n0; VAX Fileserver 63n0
		3	VAXstation 3520, 3540
		4	VAXstation 3100
		5	Reserved to DIGITAL
		6	Reserved to DIGITAL
		7	VAX 3000FT
		8–127	Reserved to DIGITAL
		128–255	Reserved to customers
11	REX520 chip		
		0	Reserved to DIGITAL
		1	VAX 4000-300
		2	VAX 6000-4n0
		3–255	Reserved to DIGITAL
12	Reserved to DIGITAL		
13	Reserved to DIGITAL		
14	VAX 9000		
15	Reserved to DIGITAL		
16	rt/uVAX (chip 78R32)		
		0	Reserved to DIGITAL
		1	rtVAX 1000
		2–127	Reserved to DIGITAL
		128–255	Reserved to customers
17	VAX 8820, 8830, 8840		
18–128	Reserved to DIGITAL		

The counter has an optional battery back-up power supply sufficient for at least 100 hours of operation, and the clock does not gain or lose any ticks during transition to or from stand-by power. The battery is recharged automatically. If the battery has failed, so that time is not accurate, then the register is cleared upon power-up. One of two things then happens:

1. The register starts counting from 0. Thus, if software initializes this clock to a value corresponding to a large time (say, a month), it can check for loss of time after a power restore by checking the clock value.

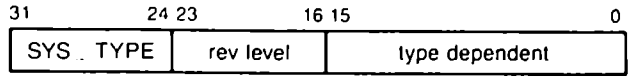


Figure 8.2
System-Type Register (SYS_TYPE)

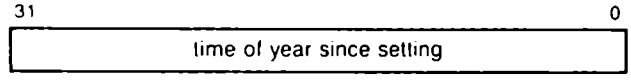


Figure 8.3
Time of Year Register (TODR)

2. The register stays at 0 until the software writes a non-zero value into it. It counts only when it contains a non-zero value.

8.3.5 *Interval Clock Registers*

The interval clock is used for accounting, for time-dependent events, and to maintain the software date and time. It provides an interrupt at IPL 22 or 24 at programmed intervals. The preferred implementation is at IPL 22. The counter is incremented at 1-microsecond intervals, with at least .01 percent accuracy (8.64 seconds per day). The clock interface consists of three internal processor registers, illustrated in Figure 8.4 and Table 8.3.

- *Interval Count Register (ICR).* The interval count register is a read-only register incremented once every microsecond. Upon a carry out (overflow) from bit ⟨31⟩, it is automatically loaded from NICR; an interrupt is generated if the interrupt is enabled. That is, the value of ICR on successive microseconds will be FFFFFFFD (hex), FFFFFFFE, FFFFFFFF, ⟨value of NICR⟩.
- *Next Interval Count Register (NICR).* This reload register is a write-only register that holds the value to be loaded into ICR when ICR overflows. The value is retained when ICR is loaded.
- *Interval Clock Control Status Register (ICCS).* The ICCS register contains control and status information for the interval clock.

Thus, to use the interval clock, load the negative of the desired interval into NICR. Then an *MTPR #^X51,#PR$_ICCS* will enable interrupts, reload ICR with the NICR interval, and set run. Every

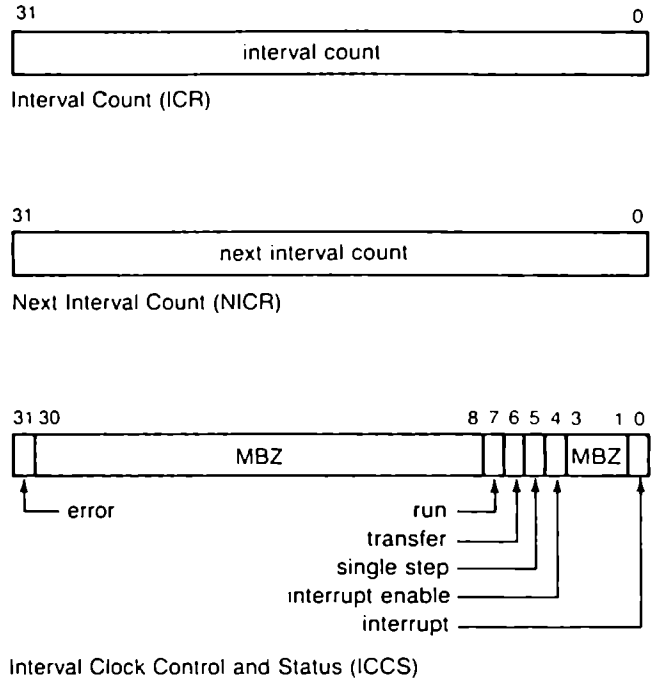


Figure 8.4
Clock Interface Internal Processor Registers

"interval count" microseconds will cause interrupt to be set and an interrupt to be requested. The interrupt routine should execute an *MTPR #^XC1,#PRS_ICCS* to clear the interrupt. If interrupt has not been cleared (the interrupt has not been handled) by the time of the next ICR overflow, error will be set.

Notes

1. If NICR is written while the clock is running, the clock may lose or add a few ticks. If the interval clock interrupt is enabled, this may cause the loss of an interrupt.
2. Processor initialization leaves ICR and NICR UNPREDICTABLE, clears ICCS ⟨6⟩ and ⟨0⟩, and leaves the rest of ICCS UNPREDICTABLE.
3. Subset processors may omit NICR and ICR and are required only to implement ICCS⟨IE⟩. If this bit is set, an interrupt request at IPL 22 is generated once every 10 milliseconds.

Table 8.3
Fields of the ICCS Register

Name	Extent	Description
error	⟨31⟩	When ICR overflows, if bit ⟨7⟩ is already set, then this bit is set. Thus, this bit indicates a missed clock tick. Write 1 to clear. Note: On a subset implementation, this bit must be zero.
interrupt	⟨7⟩	Set by hardware every time ICR overflows. If bit ⟨6⟩ is set, then an interrupt is also generated. Writing a 1 to this bit with MTPR clears it, thereby reenabling the clock tick interrupt. Note: On a subset implementation, this bit must be zero.
interrupt enable	⟨6⟩	When set, an interrupt request is generated every time ICR overflows (every time bit ⟨7⟩ is set). When clear, no interrupt is requested. Similarly, if bit ⟨7⟩ is already set and the software sets this bit, an interrupt is generated. That is, an interrupt is generated whenever the function (bit ⟨6⟩ AND bit ⟨7⟩) changes from 0 to 1. Processor initialization clears interrupt enable.
single step	⟨5⟩	If bit ⟨0⟩ is clear, each time this bit is set, ICR is incremented by 1. Write only. Note: On a subset implementation this bit must be zero.
transfer	⟨4⟩	When a 1 is written to this bit, NICR is transferred to ICR. Write only. Note: On a subset implementation this bit must be zero.
run	⟨0⟩	When set, ICR increments each microsecond. When clear, ICR does not increment automatically. Processor initialization clears this bit. Note: On a subset implementation this bit must be zero.

9

PDP-11 Compatibility Mode

Implementation of PDP-11 compatibility mode is optional. Processors that do implement compatibility mode do so as specified in this chapter. Operating system software may emulate compatibility mode on processors that omit this mode.

VAX compatibility-mode hardware, in conjunction with a compatibility-mode software executive (which runs in VAX native mode), can emulate the environment provided to user programs on a PDP-11 system. This environment does not include the following features of normal PDP-11 system operation:

- Privileged instructions such as HALT and RESET
- Special instructions such as traps and WAIT
- Access to internal processor registers such as the PSW and the console switch register
- Direct access to trap and interrupt vectors
- Direct access to I/O devices
- Interrupt servicing
- Stack overflow protection
- Alternate general register sets
- Any processor mode other than user (that is, kernel and supervisor modes are not supported) and separate I and D spaces
- Floating-point instructions

This specification is based on the behavior of all PDP-11 implementations. Compatibility-mode behavior is defined as UNPREDICTABLE where there is a difference between any two PDP-11 implementations.

9.1 General Registers and Addressing Modes

All of the PDP-11 general registers and addressing modes are provided in compatibility mode. Side effects caused by a destination address calculation have no effect on source values (except in JSR), and autoincrement modes in JMP and JSR do not affect the new PC. Side effects caused by a source address calculation, however, affect the value of a register used for destination address calculation. All PDP-11 addresses are 16 bits wide. In compatibility mode, a 16-bit PDP-11 address is zero extended to 32 bits.

The operands of some PDP-11 instructions are implied by the instruction type, whereas others are specified as part of the instruction. The different kinds of operand specifiers appearing in PDP-11 instructions are shown in Figure 9.1. Address mode operand specifiers include a 3-bit mode field, specifying one of eight modes: register, register deferred, autoincrement, autoincrement deferred, autodecrement, autodecrement deferred, index, or index deferred mode. These modes are discussed in detail in the following sections.

9.1.1 Register Mode

In register mode addressing, the operand is the contents of register n:

```
operand = Rn
```

Byte operations, except for MOVB to a register, access the low order byte, that is, bits ⟨7:0⟩. The low byte is sign extended if a register is used as the destination of a MOVB instruction. If the PC is used as the destination of a byte instruction, the result is UNPREDICTABLE.

The assembler notation for register mode is Rn.

9.1.2 Register Deferred Mode

In register deferred mode addressing, the address of the operand is the contents of register n:

```
OA = Rn
operand = (OA)
```

The assembler notation for register deferred mode is (Rn) or @Rn.

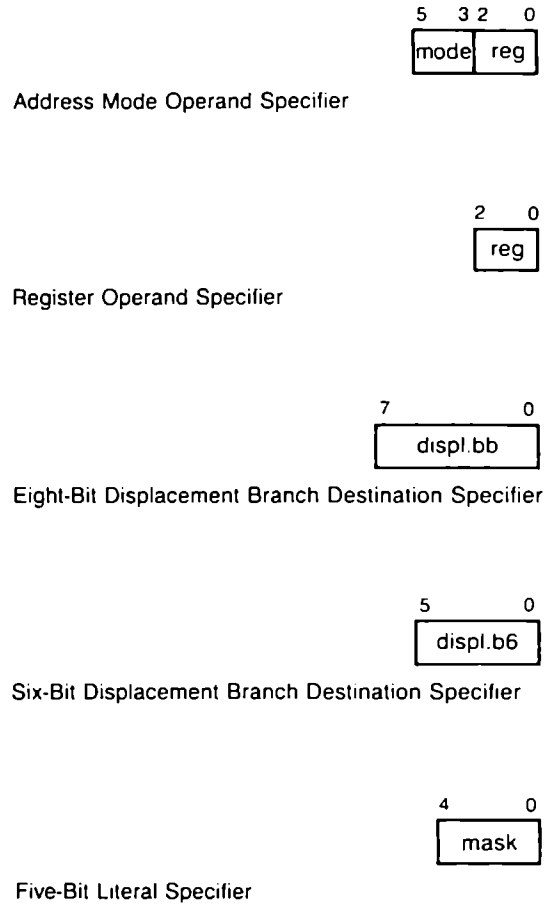


Figure 9.1
Compatibility-Mode Operand Specifiers

9.1.3 *Autoincrement Mode*

In autoincrement mode addressing, the address of the operand is the contents of register n. After the operand address is determined, the size of the operand in bytes (1 for byte, 2 for word) is added to the contents of register n (except in the case of SP and PC); the register is then replaced by the result. If Rn denotes SP or PC, the register is incremented by 2 and the register is replaced by the result.

```
OA = Rn
if n LEQ 5 then Rn ← Rn + size
else Rn ← Rn + 2
operand = (OA)
```

If Rn denotes PC, immediate data follows the instruction. The mode is termed "immediate mode."

The assembler notation for autoincrement mode is (Rn)+. For immediate mode, the notation is #constant, where constant is the data immediately following the instruction.

9.1.4 *Autoincrement Deferred Mode*

In autoincrement deferred mode addressing, the address of the operand is the contents of a word whose address is the contents of register n. After the operand address is determined, 2 is added to the contents of register n, and the register is replaced by the result.

```
OA = (Rn)
Rn ← Rn + 2
operand = (OA)
```

If Rn denotes PC, a 16-bit address follows the instruction. The mode is termed "absolute mode."

The assembler notation for autoincrement deferred mode is @(Rn)+. For absolute mode, the notation is @#address where address is the word that follows the instruction.

9.1.5 *Autodecrement Mode*

In autodecrement mode addressing, the size of the operand in bytes (1 for byte, 2 for word) is subtracted from the contents of register n (except in the case of SP and PC); the register is then replaced by the result. If Rn denotes SP or PC, the register is decremented by 2 and the register is replaced by the result. The updated contents of register n are the address of the operand:

```
if n LEQ 5 then Rn ← Rn - size
else Rn ← Rn - 2
OA = Rn
operand = (OA)
```

The assembler notation for autodecrement mode is –(Rn).

9.1.6 *Autodecrement Deferred Mode*

In autodecrement deferred mode addressing, 2 is subtracted from the contents of register n; the register is replaced by the result. The updated contents of register n are the address of the word whose contents are the address of the operand:

```
Rn ← Rn - 2
OA = (Rn)
operand = (OA)
```

The assembler notation for autodecrement deferred mode is @–(Rn).

9.1.7 Index Mode

In index mode, the index (contents of the word following the instruction) is added to the contents of register n. The result is the address of the operand:

```
OA = Rn + index
operand = (OA)
```

If Rn denotes PC, the updated contents of the PC are used. The mode is termed "relative mode."

The assembler notation for index mode is index(Rn), where the index value is the word following the instruction.

9.1.8 Index Deferred Mode

In index deferred mode, the index (contents of the word following the instruction) is added to the contents of register n. The result is the address of a word whose contents are the address of the operand:

```
OA = (Rn + index)
operand = (OA)
```

If Rn denotes PC, the updated contents of the PC are used. The mode is termed "relative deferred mode."

The assembler notation for index deferred mode is @index(Rn), where the index value is the word following the instruction.

9.2 The Stack

General register R6 is used as the stack pointer by certain instructions, as in the PDP-11 system. It is not, however, used by the hardware for any exceptions or interrupts. There is also no stack overflow protection in compatibility mode.

9.3 Processor Status Word

PDP-11 compatibility mode uses a subset of the full PDP-11 processor status word. Only bits ⟨4:0⟩ are used; bits ⟨15:5⟩ are zero. When an RTI or RTT instruction is executed, bits ⟨15:5⟩ in the saved PSW on the stack are ignored. Compatibility-mode PSW bits ⟨4:0⟩ have the same meaning as do VAX PSL bits ⟨4:0⟩. They are, respectively, PSL⟨T,N,Z,V,C⟩. See section 1.3.2 for a description of the PSL.

9.4 Instructions

Table 9.1 lists the instructions provided in compatibility mode.

Table 9.2 lists the trap instructions that cause the processor to fault to native mode, where either the complete trap may be serviced or the instruction may be simulated.

The instructions listed in Table 9.3 and all other opcodes not listed in Tables 9.1 or 9.2 are considered reserved instructions in compatibility mode. These instructions fault to native mode.

Note that no floating-point instructions are included in compatibility mode.

Figure 9.2 shows seven compatibility-mode instruction formats.

Table 9.1
Compatibility-Mode Instructions

Opcode (Octal)[a]	Mnemonic	Opcode (Octal)	Mnemonic
000002	RTI	0065SS	MFPI[b]
000006	RTT	0066DD	MTPI[b]
0001DD	JMP	1065SS	MFPD[b]
00020R	RTS	1066DD	MTPD[b]
000240–000277	Condition codes	0067DD	SXT
0003DD	SWAB	070RSS	MUL
000400–003777	Branches	071RSS	DIV
100000–103777	Branches	072RSS	ASH
004RDD	JSR	073RSS	ASHC
.050DD	CLR(B)	074RDD	XOR
.051DD	COM(B)	077RNN	SOB
.052DD	INC(B)	.1SSDD	MOV(B)
.053DD	DEC(B)	.2SSSS	CMP(B)
.054DD	NEG(B)	.3SSSS	BIT(B)
.055DD	ADC(B)	.4SSDD	BIC(B)
.056DD	SBC(B)	.5SSDD	BIS(B)
.057SS	TST(B)	06SSDD	ADD
.060DD	ROR(B)	16SSDD	SUB
.061DD	ROL(B)		
.062DD	ASR(B)		
.063DD	ASL(B)		

[a]In the opcodes, DD indicates a destination operand specifier, NN a branch displacement, R a register specifier, and SS a source operand specifier. A dot (.) indicates a 0 for word operations and a 1 for byte operations.
[b]These instructions execute exactly as they would on a PDP-11 in user mode with Instruction and Data space overmapped. More specifically, they ignore the previous access level and act like PUSH and POP instructions referencing the current stack.

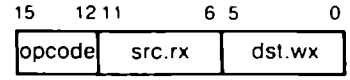
Table 9.2
Compatibility-Mode Trap Instructions

Opcode (Octal)	Mnemonic
000003	BPT
000004	IOT
104000–104377	EMT
104400–104777	TRAP

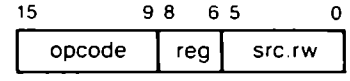
Table 9.3
Compatibility-Mode Reserved Instructions

Opcode (Octal)[a]	Mnemonic
000000	HALT
000001	WAIT
000005	RESET
000007	MFPT
00023M	SPL
0064MM	MARK
0070DD	CSM
07500R	FADD–FIS
07501R	FSUB–FIS
07502R	FMUL–FIS
07503R	FDIV–FIS
076XXX	Extended Instructions
1064SS	MTPS
1067DD	MFPS
17XXXX	FP11 Floating Point

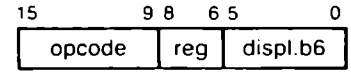
[a]In the opcodes, DD indicates a destination operand specifier, M or MM a miscellaneous field, R a register specifier, and SS a source operand specifier.



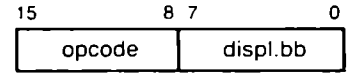
Double Operand Format with Two Address Mode Specifiers



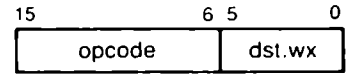
Double Operand Format with Register and Address Mode Specifiers



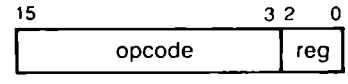
Loop Format with Register and 6-Bit Branch Displacement Specifiers



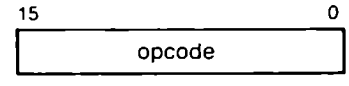
Branch Format 8-Bit Branch Displacement Specifier



Single Operand Format with Address Mode Specifier



Single Operand Format with Register Specifier



Zero Operand Format

Figure 9.2
Seven Compatibility-Mode Instruction Formats

9.4.1 Single Operand Instructions

The following single operand instructions are described in this section. The instructions are grouped according to type: arithmetic, logical, shifts, multiprecision, and rotates.

Arithmetic

- CLR(B) dst.wx
- DEC(B) dst.mx
- INC(B) dst.mx
- NEG(B) dst.mx
- TST(B) src.rx

Logical

- COM(B) dst.mx

Shifts

- ASR(B) dst.mx
- ASL(B) dst.mx

Multiprecision

- ADC(B) dst.mx
- SBC(B) dst.mx
- SXT dst.ww

Rotates

- ROL(B) dst.mx
- ROR(B) dst.mx
- SWAB dst.mw

CLR Clear

Format

```
opcode dst.wx
```

Operation

```
dst ← 0;
```

Condition codes

```
N ← 0;
Z ← 1;
V ← 0;
C ← 0;
```

Exceptions

none

Opcodes

0050	CLR	Clear Word
1050	CLRB	Clear Byte

Description

The destination operand is replaced by zero. The instruction is single operand format with address mode specifier.

DEC Decrement

Format

```
opcode dst.mx
```

Operation

```
dst ← dst - 1;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {integer overflow};
C ← C;
```

Exceptions

none

Opcodes

0053	DEC	Decrement Word
1053	DECB	Decrement Byte

Description

One is subtracted from the destination operand, and the destination operand is replaced by the result. The instruction is single operand format with address mode specifier.

Notes

1. Integer overflow occurs if the largest negative integer is decremented. On overflow, the destination operand is replaced by the largest positive integer.

INC — Increment

Format

```
opcode dst.mx
```

Operation

```
dst ← dst + 1;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {integer overflow};
C ← C;
```

Exceptions

none

Opcodes

0052	INC	Increment Word
1052	INCB	Increment Byte

Description

One is added to the destination operand, and the destination operand is replaced by the result. The instruction is single operand format with address mode specifier.

Notes

1. Integer overflow occurs if the largest positive integer is incremented. On overflow, the destination operand is replaced by the largest negative integer.

NEG — Negate

Format

```
opcode dst.mx
```

Operation

```
dst ← -dst;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← dst EQL most negative integer;
C ← dst NEQ 0;
```

Exceptions

none

Opcodes

0054	NEG	Negate Word
1054	NEGB	Negate Byte

Description

The destination operand is negated (two's complement), and the destination operand is replaced by the result. The instruction is single operand format with address mode specifier.

Notes

1. Integer overflow occurs if the operand is the most negative integer (which has no positive counterpart). On overflow, the destination operand is replaced by itself.

TST Test

Format

```
opcode src.rx
```

Operation

```
src - 0;
```

Condition codes

```
N ← src LSS 0;
Z ← src EQL 0;
V ← 0;
C ← 0;
```

Exceptions

none

Opcodes

0057	TST	Test Word
1057	TSTB	Test Byte

Description

The condition codes are affected according to the value of the source operand. The instruction is single operand format with address mode specifier.

COM Complement

Format

```
opcode dst.mx
```

Operation

```
dst ← NOT dst;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← 0;
C ← 1;
```

Exceptions

none

Opcodes

0051	COM	Complement Word
1051	COMB	Complement Byte

Description

The destination operand is complemented (one's complement), and the destination operand is replaced by the result. The instruction is single operand format with address mode specifier.

ASR — Arithmetic Shift Right

Format

```
opcode dst.mx
```

Operation

```
dst ← dst shifted one place to the right;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {bit shifted out} XOR {dst LSS 0};
C ← bit shifted out;
```

Exceptions

none

Opcodes

0062	ASR	Arithmetic Shift Right Word
1062	ASRB	Arithmetic Shift Right Byte

Description

The destination operand is arithmetically shifted right by 1 bit, and the destination operand is replaced by the result. The instruction is single operand format with address mode specifier.

Notes

1. The sign bit of the destination operand is replicated in shifts to the right. The condition code C-bit stores the bit shifted out.
2. If the PC is used as the destination operand, the result and the next instruction executed are UNPREDICTABLE.

ASL — Arithmetic Shift Left

Format

```
opcode dst.mx
```

Operation

```
dst ← dst shifted one place to the left;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {integer overflow};
C ← bit shifted out;
```

Exceptions

none

Opcodes

0063	ASL	Arithmetic Shift Left Word
1063	ASLB	Arithmetic Shift Left Byte

Description

The destination operand is arithmetically shifted left by 1 bit, and the destination operand is replaced by the result. The instruction is single operand format with address mode specifier.

Notes

1. The least significant bit is filled with zero in shifts to the left. The condition code C-bit stores the bit shifted out.
2. Integer overflow occurs if the destination changes sign due to the shift.

ADC — Add Carry

Format

```
opcode dst.mx
```

Operation

```
dst ← dst + C;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {integer overflow};
C ← {carry from most significant bit};
```

Exceptions

none

Opcodes

0055	ADC	Add Carry to Word
1055	ADCB	Add Carry to Byte

Description

The contents of the condition code C-bit are added to the destination operand, and the destination operand is replaced by the result. The instruction is single operand format with address mode specifier.

Notes

1. Integer overflow occurs if the most positive integer is incremented. On overflow, the result is the most negative integer.

SBC — Subtract Carry

Format

```
opcode dst.mx
```

Operation

```
dst ← dst - C;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {integer overflow};
C ← {borrow into most significant bit};
```

Exceptions

none

Opcodes

0056	SBC	Subtract Carry from Word
1056	SBCB	Subtract Carry from Byte

Description

The contents of the condition code C-bit are subtracted from the destination operand, and the destination operand is replaced by the result. The instruction is single operand format with address mode specifier.

Notes

1. Integer overflow occurs if the most negative integer is decremented. On overflow, the result is the most positive integer.

SXT — Sign Extend Word

Format

```
opcode dst.ww
```

Operation

```
if N EQL 1 then dst ← -1 else dst ← 0;
```

Condition codes

```
N ← dst LSS 0;          !N ← N
Z ← dst EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

0067 SXT Sign Extend

Description

If the condition code N-bit is set, then the destination operand is replaced by –1; otherwise, the destination operand is cleared. The instruction is single operand format with address mode specifier.

Notes

1. If the PC is used as the destination operand, the results and the next instruction executed are UNPREDICTABLE.

ROL — Rotate Left

Format

```
opcode dst.mx
```

Operation

```
dst'C ← dst'C rotated left;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {integer overflow};
C ← {bit rotated out of dst};
```

Exceptions

none

Opcodes

0061	ROL	Rotate Left Word
1061	ROLB	Rotate Left Byte

Description

The condition code C-bit and the destination operand are rotated left by one bit position; that is, the C-bit gets the most significant bit of the destination operand, and the destination is replaced by the destination shifted left by one bit with the initial C-bit filling the least significant bit. The instruction is single operand format with address mode specifier.

Notes

1. The rotate instructions operate on the destination operand and the condition code C-bit taken as a circular datum.
2. Integer overflow occurs if the destination changes sign because of the rotate.

ROR — Rotate Right

Format

```
opcode dst.mx
```

Operation

```
dst'C ← dst'C rotated right;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {C bit changed due to rotate};
C ← {bit rotated out of dst};
```

Exceptions

none

Opcodes

0060	ROR	Rotate Right Word
1060	RORB	Rotate Right Byte

Description

The condition code C-bit and the destination operand are rotated right by one bit position; that is, the C-bit gets the least significant bit of the destination operand, and the destination is replaced by the destination shifted right by one bit with the initial C-bit filling the most significant bit. The instruction is single operand format with address mode specifier.

Notes

1. The rotate instructions operate on the destination operand and the condition code C-bit taken as a circular datum.

SWAB — Swap Bytes

Format

```
opcode dst.mw
```

Operation

```
dst ← dst<7:0>'dst<15:8>;
```

Condition codes

```
N ← dst<7:0> LSS 0;
Z ← dst<7:0> EQL 0;
V ← 0;
C ← 0;
```

Exceptions

none

Opcodes

0003	SWAB	Swap Bytes

Description

The high and low bytes of the destination word operand are swapped. The instruction is single operand format with address mode specifier.

Notes

1. If the PC is used as the destination operand, the result and the next instruction executed are UNPREDICTABLE.

9.4.2 Double Operand Instructions

PDP-11 compatibility-mode double-operand instructions are described in this section. The instructions are grouped according to type: arithmetic and logical, and shift.

Arithmetic and Logical

- MOV(B) src.rx, dst.mx
- ADD src.rw, dst.mw
- SUB src.rw, dst.mw
- CMP(B) src1.rx, src2.rx
- MUL reg, src.rw
- DIV reg, src.rw
- XOR reg, dst.mw
- BIS(B) src.rx, dst.mx
- BIC(B) src.rx, dst.mx
- BIT(B) src1.rx, src2.rx

Shift

- ASH reg, src.rw
- ASHC reg, src.rw

If a register that is used in the source operand specifier in autoincrement or autodecrement modes is also used in the destination (or source 2) operand specifier, the updated value of the register is used to evaluate the destination specifier. Side effects caused by a destination address calculation have no effect on source values.

MOV — Move

Format

```
opcode src.rx, dst.wx
```

Operation

```
dst ← src;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

01	MOV	Move Word
11	MOVB	Move Byte

Description

The destination operand is replaced by the source operand. The instruction is double operand format with two address mode specifiers.

Notes

1. The low byte is sign extended on a MOVB to a register; that is, bits ⟨15:8⟩ of the destination register are replaced by bit ⟨7⟩ of the source operand.

ADD — Add

Format

```
opcode src.rw, dst.mw
```

Operation

```
dst ← dst + src;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {integer overflow};
C ← {carry from most significant digit};
```

Exceptions

none

Opcodes

06	ADD	Add Word

Description

The source operand is added to the destination operand, and the destination operand is replaced by the result. The instruction is double operand format with two address mode specifiers.

Notes

1. Integer overflow occurs if the input operands have the same sign and the result has the opposite sign. On overflow, the destination operand is replaced by the low-order bits of the true result.

SUB — Subtract

Format

```
opcode src.rw, dst.mw
```

Operation

```
dst ← dst - src;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {integer overflow};
C ← {borrow into most significant digit};
```

Exceptions

none

Opcodes

16	SUB	Subtract Word

Description

The source operand is subtracted from the destination operand, and the destination operand is replaced by the result. The instruction is double operand format with two address mode specifiers.

Notes

1. Integer overflow occurs if the input operands are of different signs and the result has the sign of the source. On overflow, the destination operand is replaced by the low-order bits of the true result.

CMP — Compare

Format

```
opcode src1.rx, src2.rx
```

Operation

```
tmp ← src1 - src2;
```

Condition codes

```
N ← tmp LSS 0;
Z ← tmp EQL 0;
V ← {integer overflow};
C ← {borrow into most significant digit};
```

Exceptions

none

Opcodes

02	CMP	Compare Word
12	CMPB	Compare Byte

Description

The source 1 operand is compared with the source 2 operand. The only action is to set the condition codes. The instruction is double operand format with two address mode specifiers.

Notes

1. Integer overflow occurs if the operands are of different signs and the result of the subtraction (src1 – src2) has the same sign as the source 2 operand.

MOV — Move

Format

```
opcode src.rx, dst.wx
```

Operation

```
dst ← src;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

01	MOV	Move Word
11	MOVB	Move Byte

Description

The destination operand is replaced by the source operand. The instruction is double operand format with two address mode specifiers.

Notes

1. The low byte is sign extended on a MOVB to a register; that is, bits ⟨15:8⟩ of the destination register are replaced by bit ⟨7⟩ of the source operand.

ADD — Add

Format

```
opcode src.rw, dst.mw
```

Operation

```
dst ← dst + src;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {integer overflow};
C ← {carry from most significant digit};
```

Exceptions

none

Opcodes

06	ADD	Add Word

Description

The source operand is added to the destination operand, and the destination operand is replaced by the result. The instruction is double operand format with two address mode specifiers.

Notes

1. Integer overflow occurs if the input operands have the same sign and the result has the opposite sign. On overflow, the destination operand is replaced by the low-order bits of the true result.

SUB — Subtract

Format

```
opcode src.rw, dst.mw
```

Operation

```
dst ← dst - src;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← {integer overflow};
C ← {borrow into most significant digit};
```

Exceptions

none

Opcodes

16	SUB	Subtract Word

Description

The source operand is subtracted from the destination operand, and the destination operand is replaced by the result. The instruction is double operand format with two address mode specifiers.

Notes

1. Integer overflow occurs if the input operands are of different signs and the result has the sign of the source. On overflow, the destination operand is replaced by the low-order bits of the true result.

CMP — Compare

Format

```
opcode src1.rx, src2.rx
```

Operation

```
tmp ← src1 - src2;
```

Condition codes

```
N ← tmp LSS 0;
Z ← tmp EQL 0;
V ← {integer overflow};
C ← {borrow into most significant digit};
```

Exceptions

none

Opcodes

02	CMP	Compare Word
12	CMPB	Compare Byte

Description

The source 1 operand is compared with the source 2 operand. The only action is to set the condition codes. The instruction is double operand format with two address mode specifiers.

Notes

1. Integer overflow occurs if the operands are of different signs and the result of the subtraction (src1 – src2) has the same sign as the source 2 operand.

MUL Multiply

Format

```
opcode reg, src.rw
```

Operation

```
tmp<31:0> ← Rn * src;
Rn ← tmp<31:16>;
R[n OR 1] ← tmp<15:0>;
```

Condition codes

```
N ← tmp LSS 0;
Z ← tmp EQL 0;
V ← 0;
C ← {result cannot be represented in 16 bits};
```

Exceptions

none

Opcodes

070 MUL Multiply Word

Description

The destination register is multiplied by the source operand. The most significant 16 bits of the 32-bit product are stored in register Rn. Then the least significant 16 bits are stored in R[n OR 1]. The condition codes are set based on the 32-bit result. The instruction is double operand format with register and address mode specifiers.

Notes

1. The C-bit is set if the result of the multiplication cannot be represented in 16 bits; that is, if the 32-bit product is less than -2^{15} or greater than or equal to 2^{15}.
2. If an odd-numbered register is used as the destination, the low-order 16 bits are stored as the result.
3. If R6 or PC is used as the destination, the next instruction executed and the result are UNPREDICTABLE.

DIV Divide

Format

```
opcode reg, src.rw
```

Operation

```
tmp ← Rn'R[n OR 1]
Rn ← tmp / src;
R[n OR 1] ← REM(tmp, src);
```

Condition codes

```
N ← Rn LSS 0;          !UNPREDICTABLE if V is set
Z ← Rn EQL 0;          !UNPREDICTABLE if V is set
V ← {src EQL 0} OR {integer overflow};
C ← {src EQL 0};
```

Exceptions

none

Opcodes

071 DIV Divide

Description

If the source operand is not zero, the 32-bit integer in Rn'R[n OR 1] is divided by the source operand. The quotient is stored in Rn, and the remainder is stored in R[n OR 1]. The remainder has the same sign as the dividend. If the source operand is zero, the instruction terminates without modifying the destination registers.

Notes

1. Integer overflow occurs if the quotient is less than -2^{15} or greater than or equal to 2^{15}. On integer overflow, the contents of the destination registers are UNPREDICTABLE.
2. If an odd register or R6 is used as the destination, the results are UNPREDICTABLE. Furthermore, if R6 or PC is used as the destination, the next instruction executed is UNPREDICTABLE.

XOR — Exclusive-OR

Format

```
opcode reg, dst.mw
```

Operation

```
dst ← Rn XOR dst;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

074 XOR Exclusive-OR Word

Description

The source register is XORed with the destination operand, and the destination operand is replaced by the result. The instruction is double operand format with register and address mode specifiers.

BIS — Bit Set

Format

```
opcode src.rx, dst.mx
```

Operation

```
dst ← dst OR src;
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

05 BIS Bit Set Word
15 BISB Bit Set Byte

Description

The source operand is ORed with the destination operand, and the destination operand is replaced by the result. The instruction is double operand format with two address mode specifiers.

BIC — Bit Clear

Format

```
opcode src.rx, dst.mx
```

Operation

```
dst ← dst AND {NOT src};
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

04	BIC	Bit Clear Word
14	BICB	Bit Clear Byte

Description

The destination operand is ANDed with the one's complement of the source operand, and the destination operand is replaced by the result. The instruction is double operand format with two address mode specifiers.

BIT — Bit Test

Format

```
opcode src1.rx, src2.rx
```

Operation

```
tmp ← src1 AND src2;
```

Condition codes

```
N ← tmp LSS 0;
Z ← tmp EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

03	BIT	Bit Test Word
13	BITB	Bit Test Byte

Description

The source 1 operand is ANDed with the source 2 operand. The only action is to set the condition codes. The instruction is double operand format with two address mode specifiers.

ASH Arithmetic Shift

Format

```
opcode reg, src.rw
```

Operation

```
Rn ← Rn shifted src<5:0> bits;
```

Condition codes

```
N ← Rn LSS 0;
Z ← Rn EQL 0;
V ← if src<5:0> EQL 0 then 0 else {integer overflow};
C ← if src<5:0> EQL 0 then 0 else {last bit shifted out};
```

Exceptions

none

Opcodes

072 ASH Arithmetic Shift

Description

The specified register is arithmetically shifted by the number of bits specified by the count operand (bits ⟨5:0⟩ of the source operand), and the register is replaced by the result. The count ranges from –32 to +31. A negative count signifies a right shift. A positive count signifies a left shift. A zero count implies no shift, but condition codes are affected. The instruction is double operand format with register and address mode specifiers.

Notes

1. The sign bit of Rn is replicated in shifts to the right. The least significant bit is filled with zero in shifts to the left. The C-bit stores the last bit shifted out.
2. Integer overflow occurs on a left shift if any bit shifted into the sign position differs from the initial sign bit of the register.
3. If the PC is used as the destination operand, the result and the next instruction executed are UNPREDICTABLE.

ASHC Arithmetic Shift Combined

Format

```
opcode reg, src.rw
```

Operation

```
tmp ← Rn'R[n OR 1];
tmp ← tmp shifted src<5:0> bits;
Rn ← tmp<31:16>;
R[n OR 1] ← tmp<15:0>;
```

Condition codes

```
N ← tmp LSS 0;
Z ← tmp EQL 0;
V ← if src<5:0> EQL 0 then 0 else {integer overflow};
C ← if src<5:0> EQL 0 then 0 else {last bit shifted out};
```

Exceptions

none

Opcodes

073 ASHC Arithmetic Shift Combined

Description

The contents of the specified register, Rn, and a second register R[n OR 1] are treated as a single 32-bit operand and are shifted by the number of bits specified by the count operand (bits ⟨5:0⟩ of the source operand); the registers are replaced by the result. First, bits ⟨31:16⟩ of the result are stored in register Rn. Then, bits ⟨15:0⟩ of the result are stored in register R[n OR 1]. The count ranges from –32 to +31. A negative count signifies a right shift. A positive count signifies a left shift. A zero count implies no shift, but condition codes are affected. Condition codes are always set on the 32-bit result. The instruction is double operand format with register and address mode specifiers.

Notes

1. The sign bit of Rn is replicated in shifts to the right. The least significant bit is filled with zero in shifts to the left. The C-bit stores the last bit shifted out.
2. Integer overflow occurs on a left shift if any bit shifted into the sign position differs from the initial sign bit of the 32-bit operand.
3. If the SP or PC is used as the destination operand, the result and the next instruction executed are UNPREDICTABLE.

9.4.3 *Branch Instructions*

The following PDP-11 compatibility-mode branch instructions are described in this section.

- BCC displ.bb
- BCS displ.bb
- BEQ displ.bb
- BGE displ.bb
- BGT displ.bb
- BHI displ.bb
- BHIS displ.bb
- BLE displ.bb
- BLO displ.bb
- BLOS displ.bb
- BLT displ.bb
- BMI displ.bb
- BNE displ.bb
- BPL displ.bb
- BR displ.bb
- BVC displ.bb
- BVS displ.bb
- SOB reg, displ.b6

BR Branch

Format `opcode displ.bb`

Operation `PC ← PC + SEXT(2*displ);`

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions none

Opcodes 0004 BR Branch

Description Twice the sign-extended displacement is added to the PC, and the PC is replaced by the result. The instruction is branch format with 8-bit displacement.

B Branch on (condition)

Format

```
opcode displ.bb
```

Operation

```
if condition then PC ← PC + SEXT(2*displ);
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

none

Opcodes

0014	BEQ	Z EQL 1	Branch on Equal
0010	BNE	Z EQL 0	Branch Not Equal
1004	BMI	N EQL 1	Branch on Minus
1000	BPL	N EQL 0	Branch on Plus
1034	BCS	C EQL 1	Branch on Carry Set
	BLO		Branch on Lower
1030	BCC	C EQL 0	Branch on Carry Clear
	BHIS		Branch on Higher or Same
1024	BVS	V EQL 1	Branch on Overflow Set
1020	BVC	V EQL 0	Branch on Overflow Clear
0024	BLT	{N XOR V} EQL 1	Branch on Less Than
0020	BGE	{N XOR V} EQL 0	Branch on Greater Than or Equal
0034	BLE	{Z OR {N XOR V}} EQL 1	Branch on Less Than or Equal
0030	BGT	{Z OR {N XOR V}} EQL 0	Branch on Greater Than
1010	BHI	{C OR Z} EQL 0	Branch on Higher
1014	BLOS	{C OR Z} EQL 1	Branch on Lower or Same

Description

The condition codes are tested, and, if the condition indicated by the instruction is met, twice the sign-extended displacement is added to the PC; the PC is replaced by the result. These instructions are branch format with 8-bit displacement.

SOB — Subtract One and Branch

Format

```
opcode reg, displ.b6
```

Operation

```
Rn ← Rn - 1;
if Rn NEQ 0 then PC ← PC - ZEXT(2*displ);
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

none

Opcodes

077 SOB Subtract One and Branch

Description

One is subtracted from the specified register, and the register is replaced by the result. If the register is not equal to zero, twice the zero-extended displacement is subtracted from the PC; the PC is replaced by the result. The instruction is loop format.

Notes

1. If the PC is specified as the register, the results and the next instruction executed are UNPREDICTABLE.
2. The 6-bit displacement operand is contained in bits ⟨5:0⟩ of the instruction.

9.4.4 *Jump and Subroutine Instructions*

The following PDP-11 compatibility-mode jump and subroutine instructions are described in this section.

- JMP dst.aw
- JSR reg, dst.aw
- RTS reg

JMP Jump

Format

```
opcode dst.aw
```

Operation

```
PC ← dst;
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

```
compatibility-mode illegal instruction
```

Opcodes

0001 JMP Jump

Description

The PC is replaced by the destination operand. The instruction is single operand format with address mode specifier.

Notes

1. A compatibility-mode illegal instruction fault occurs if destination mode 0 is used.

JSR — Jump to Subroutine

Format

```
opcode reg, dst.aw
```

Operation

```
tmp ← dst;          ! Value of Rn is affected by
-(SP) ← Rn;         ! dst specifier evaluation.
Rn ← PC;
PC ← tmp;
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

```
compatibility-mode illegal instruction
```

Opcodes

004 JSR Jump to Subroutine

Description

The source register is pushed on the stack, and the source register is replaced by the PC. The PC is replaced by the destination operand. The instruction is double operand format with register and address mode specifier.

Notes

1. A compatibility-mode illegal instruction fault occurs if destination mode 0 is used.
2. If the destination uses the same register as the source in the autoincrement or autodecrement addressing modes, the updated contents of the register are pushed on the stack.

RTS — Return from Subroutine

Format

```
opcode reg
```

Operation

```
PC ← Rn;
Rn ← (SP)+;
```

Condition codes

```
N ← N;
Z ← Z;
V ← V;
C ← C;
```

Exceptions

```
none
```

Opcodes

00020 RTS Return from Subroutine

Description

The PC is replaced by the destination register. The destination register is replaced by a word popped from the stack. The instruction is single operand format with register specifier.

9.4.5 *Return from Interrupts and Traps*

The following PDP-11 compatibility-mode return-from-interrupts and return-from-trap instructions are described in this section.

- RTI
- RTT

RTx Return from Interrupt/Trap

Format

```
opcode
```

Operation

```
PC ← (SP)+;
PSW<4:0> ← {(SP)+}<4:0>;
```

Condition codes

```
N ← saved PSW<3>;
Z ← saved PSW<2>;
V ← saved PSW<1>;
C ← saved PSW<0>;
```

Exceptions

none

Opcodes

000002	RTI	Return from Interrupt
000006	RTT	Return from Trap

Description

The PC is replaced by the first word popped from the stack. The low five bits of the PSW are replaced by the corresponding bits of the second word popped from the stack. The instruction is zero operand format.

Notes

1. In PDP-11 compatibility mode, the RTI and RTT instructions ignore the high 11 bits of the PSW popped from the stack.
2. In PDP-11 compatibility mode, the RTI and RTT instructions are identical.

9.4.6 *Miscellaneous Instructions*

The following miscellaneous compatibility-mode instructions are described in this section.

- MTPI,D dst.ww
- MFPI,D src.rw
- NOP
- CLC
- CLV
- CLZ
- CLN
- CCC
- SEC
- SEV
- SEZ
- SEN
- SCC

MTP — Move To Previous Space

Format

```
opcode dst.ww
```

Operation

```
tmp ← (SP)+;    !Pop source from stack (updating SP)
dst ← tmp;      !Write source to destination
```

Condition codes

```
N ← dst LSS 0;
Z ← dst EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

0066	MTPI	Move To Previous Instruction Space
1066	MTPD	Move To Previous Data Space

Description

In compatibility mode, this PDP-11 instruction works like a POP instruction. The destination operand is replaced by a word popped from the stack. The instruction is single operand format with address mode specifier.

Notes

1. The implied source operand specifier is evaluated before the destination specifier.

MFP — Move From Previous Space

Format

```
opcode src.rw
```

Operation

```
-(SP) ← src;
```

Condition codes

```
N ← src LSS 0;
Z ← src EQL 0;
V ← 0;
C ← C;
```

Exceptions

none

Opcodes

0065	MFPI	Move From Previous Instruction Space
1065	MFPD	Move From Previous Data Space

Description

In compatibility mode, this PDP-11 instruction works like a PUSH instruction. The source operand is pushed on the stack. The instruction is single operand format with address mode specifier.

CC Condition Code Operators

Format

```
opcode mask
```

Operation

```
if mask<4> EQL 1 then PSW<3:0> ← PSW<3:0> OR mask<3:0>
else PSW<3:0> ← PSW<3:0> AND {NOT mask<3:0>};
```

Condition codes

```
if mask<4> EQL 1 then
   begin
   N ← N OR mask<3>;
   Z ← Z OR mask<2>;
   V ← V OR mask<1>;
   C ← C OR mask<0>;
   end
else
   begin
   N ← N AND {NOT mask<3>};
   Z ← Z AND {NOT mask<2>};
   V ← V AND {NOT mask<1>};
   C ← C AND {NOT mask<0>};
   end
```

Exceptions

none

Opcodes

000240	NOP	No operation
000241	CLC	Clear C
000242	CLV	Clear V
000244	CLZ	Clear Z
000250	CLN	Clear N
000257	CCC	Clear all Condition Codes
000261	SEC	Set C
000262	SEV	Set V
000264	SEZ	Set Z
000270	SEN	Set N
000277	SCC	Set all Condition Codes

Combinations of the above set or clear operations may be ORed together to form combined instructions.

Description

If the mask⟨4⟩ bit is set, the PSW condition code bits are ORed with mask⟨3:0⟩ and the condition codes are replaced by the result. If the mask⟨4⟩ bit is clear, the PSW condition code bits are ANDed with the one's complement of mask⟨3:0⟩ and the condition codes are replaced by the result. The instruction is zero operand format. Bits ⟨4:0⟩ of the opcode are used as the mask operand.

9.5 Entering and Leaving Compatibility Mode

Compatibility mode is entered by executing an REI instruction with the compatibility-mode bit set in the PSL on the stack. Other bits in the PSL either have the effects they have in native mode or are required to have specific values in compatibility mode. PSL⟨TP⟩, ⟨T⟩, ⟨N⟩, ⟨Z⟩, ⟨V⟩, and ⟨C⟩ have the same effects and meanings as they have in native mode. PSL⟨FPD⟩, ⟨IS⟩, ⟨IPL⟩, ⟨IV⟩, ⟨FU⟩, ⟨DV⟩ must be 0, and ⟨CUR_MOD⟩ and ⟨PRV_MOD⟩ must be 3.

VAX native mode is returned to from compatibility mode by the compatibility-mode program causing an exception; or by an interrupt.

Note that when an RTI or RTT instruction is executed in compatibility mode, the 11 high bits of the PSW are ignored. But when the PSW is restored as part of the PSL when going from VAX native mode to compatibility mode, those bits must be 0, or a reserved operand fault occurs.

9.6 Native-Mode and Compatibility-Mode Registers

Compatibility-mode registers R0 through R6 are bits ⟨15:0⟩ of VAX general registers R0 through R6, respectively. Compatibility-mode register R7 (PC) is bits ⟨15:0⟩ of VAX general register R15 (PC). VAX registers R8 through R14 (SP) are not affected by compatibility mode. When entering compatibility mode, VAX register R7 and the upper halves of registers R0 through R6 and R15 are ignored. When an exception or interrupt occurs from compatibility mode, VAX register R7 is UNPREDICTABLE and the upper halves of R0 through R6 are either cleared or left unchanged; the upper half of the stacked R15 (PC) is zero. Since there are no FP11 floating-point instructions in compatibility mode, there are no floating accumulators.

9.7 Compatibility-Mode Memory Management

PDP-11 addresses are 16-bit byte addresses. Hence, compatibility-mode programs are confined to execute in the first 64K bytes of the per-process part of the virtual address space. A one-to-one correspondence exists between a compatibility-mode virtual

address and its VAX counterpart. (Virtual address 0, for example, references the same location in both modes.) A compatibility mode address is interpreted in the following paragraphs as a native mode address by appending zero in bits ⟨31:16⟩ to the compatibility-mode address in bits ⟨15:0⟩.

PDP-11 segments can consist of 1 to 128 blocks of 64 bytes. VAX pages are 512 bytes long. The PDP-11 capability of providing different access protection to different segments is provided in 8-block chunks, since protection is specified at the page level in the VAX architecture.

The memory management system protects and relocates compatibility-mode addresses in the normal native mode manner. Thus, all of the memory management mechanisms available in native mode are available to the compatibility-mode executive for managing both the virtual and the physical memory of compatibility-mode programs. All of the exception conditions that can be caused by memory management in native mode can also occur when relocating a compatibility-mode address. See Chapter 4.

9.8 Compatibility-Mode Exceptions and Interrupts

All interrupts and exception conditions that occur while the processor is in compatibility mode cause the processor to enter native mode. These conditions are serviced as indicated in Chapter 5 (note that this includes backing up instruction side effects if necessary). The exception conditions discussed in this section are specific to compatibility mode. All these exceptions create a three-longword frame on the kernel stack containing PSL and PC, and one longword of exception-dependent information. Bits ⟨15⟩ through ⟨0⟩ of this longword contain a code indicating the specific type of exception, and bits ⟨31⟩ through ⟨16⟩ are zero. There are no compatibility-mode exception conditions that result in traps. (See Chapter 5 for definitions of trap, fault, and abort.)

9.8.1 Odd Address Error Abort

An odd address error abort is caused in compatibility mode whenever a word reference is attempted on a byte boundary. The code for odd address errors is 6.

9.8.2 Faults

Compatibility-mode instruction faults and their corresponding code numbers are as follows:

- *Reserved Instruction Fault.* A reserved instruction fault occurs for opcodes that are defined as reserved in compatibility mode (see section 9.4 earlier in this chapter). The code for the reserved instruction fault is 0.
- *BPT Instruction Fault.* The code for the BPT instruction fault is 1.
- *IOT Instruction Fault.* The code for the IOT instruction fault is 2.
- *EMT Instruction Fault.* The fault code for the group of EMT instructions is 3.
- *TRAP Instruction Fault.* The fault code for the group of TRAP instructions is 4.
- *Illegal Instruction Fault.* In compatibility mode, JMP and JSR instructions with a register destination are illegal. The fault code for illegal instructions is 5.

9.9 Tracing in Compatibility Mode

In compatibility mode, a trace fault occurs at the beginning of an instruction when PSW⟨T⟩ is set at the beginning of the prior instruction. This effect is achieved by using PSL⟨TP⟩. (See section 5.7.5 for a description of tracing.) On trace faults, a two-longword kernel stack frame is created, containing PSL and PC. IPL and IS are 0, and CM is 1 in the stacked PSL. Compatibility-mode trace fault uses the same vector as native-mode trace fault (see Chapter 5). The rules for trace fault generation in compatibility mode are identical to those for native mode. However, an odd address abort for an instruction fetch may precede the trace fault for that instruction.

There are two ways to get PSW⟨T⟩ set at the beginning of a compatibility-mode instruction:

- An RTT or RTI instruction is executed in compatibility mode with T set in the PSW image on the stack. In this case, the next instruction is executed (the 1 pointed to by the PC on the stack), and a trace fault is taken before the following instruction.

- An REI instruction is executed in native mode that has both T and CM bit set (and TP clear) in the saved PSL image on the stack. Again, one instruction is executed, and the trace fault is taken. (See Chapter 5 for a complete description of the interaction of REI, PSL⟨T⟩, and PSL⟨TP⟩. The operations that occur as a function of these conditions are the same whether or not compatibility mode is being entered from the REI.)

PSW⟨T⟩ interacts with other compatibility-mode operations as follows. For interaction with other than compatibility mode, see Chapter 5.

1. PSW⟨T⟩ is set (but PSL⟨TP⟩ is clear) at the beginning of any compatibility-mode instruction that does not cause a compatibility-mode fault. In this case, the instruction sets PSL⟨TP⟩ and executes. A trace fault is taken before the next instruction. The saved PSL has T set and TP clear. The compatibility-mode executive can take one of the following courses of action:
   a. If it services the exception directly, it can clear T in the saved PSL on the kernel stack if it no longer wants to trace the program; or it can leave T set if it wants to continue tracing the program. It exits with an REI.
   b. If it returns the trace exception to compatibility mode, it pushes a (16-bit) PC and (16-bit) PSW with T set on the compatibility-mode user stack to simulate the effect of the PDP-11 trace trap. It then clears T in the saved PSL image on the kernel stack, changes the saved PC to point to the compatibility-mode service routine, and does an REI. The compatibility-mode service routine can then clear T in the PSW image on its stack if it does not want to continue tracing. The compatibility-mode routine returns with RTT or RTI.
2. PSW⟨T⟩ is set (but PSL⟨TP⟩ is clear) at the beginning of an RTI or RTT. The RTT or RTI instruction executes, and PSL⟨TP⟩ is set. A trace fault occurs before the next instruction is executed. Two different cases exist depending on whether or not T was set in the image of the PSW that was popped from the stack by the RTT or RTI instruction:
   a. PSW⟨T⟩ is not set. Neither TP nor T will be set in the saved PSL on the kernel stack.
   b. PSW⟨T⟩ is set. PSL⟨TP⟩ will not be set, and PSW⟨T⟩ will be set, as is the case for other compatibility-mode instructions.

3. PSW⟨T⟩ is set (but PSL⟨TP⟩ is clear) at the beginning of any instruction that causes a compatibility mode fault. The fault condition is serviced first. TP is clear and T is set in the saved PSL pushed on the kernel stack.

9.10 Unimplemented PDP-11 Traps

Several traps that occur in PDP-11 systems are not implemented in compatibility mode:

- There is no stack overflow trap. This is equivalent to the user mode of the KT11 where there is also no overflow protection. Stack overflow can be provided by the compatibility-mode executive using the memory management mechanisms.
- There is no concept of a double error trap in compatibility mode, since the first error always puts the processor in native mode.
- All other exception conditions, such as power failure, memory parity, and memory management exceptions, cause the processor to enter native mode.

9.11 Compatibility-Mode I/O References

Instruction stream references, data reads, and data writes cannot be to I/O space. The results are UNPREDICTABLE if I/O space is referenced from compatibility mode.

9.12 Processor Registers

The only processor register available in compatibility mode is part of the PSW, and it may be explicitly referenced only with the condition code instructions, RTI, and RTT. Access to all other registers must be done in native mode.

9.13 Program Synchronization

All PDP-11 systems guarantee that read-modify-write operations to I/O device registers are interlocked; that is, the device can determine at the time of the read that the same register will be written as the next bus cycle. This synchronization also works

in memory on most PDP-11 systems. In compatibility mode, instructions that have modify destinations will perform this synchronization for UNIBUS I/O device registers, if the system has a UNIBUS interconnect, and never for memory.

Compatibility-mode procedures can write data that is to be subsequently executed as an instruction without requiring any additional synchronization.

10

System Bootstrapping and Console

A VAX processor can be in one of five major states: attempting to load and start (bootstrap) the operating system, attempting to restart the operating system, powered off, halted, or running. This chapter describes the processor when it is not running and describes the transitions between major states.

The four major states described in this chapter are differentiated from the running state. When the processor is running, it interprets instructions, services interrupts and exceptions, and initiates I/O operations. The console acts like a normal operating system terminal (the console is in program I/O mode).

When the processor is halted, it does not interpret instructions, service interrupts or exceptions, or initiate I/O operations. The console interprets a command language that provides control over the system (the console is in console I/O mode).

When system power supplies are unable to provide power to the processor, the processor halts, and is powered off.

The console can restart a halted operating system and can also load and start (bootstrap) an operating system. How the console handles these states is described in the following sections.

10.1 System Bootstrapping

System bootstraping can occur as the result of a powerfail recovery, a processor halt, or the operator entering a BOOT command

at the console. Also, a bootstrap can occur if the operating system moves a value of 0F02 into the TXDB register. See section 10.6.1 for more information about TXDB. See section 10.4 for a complete description of these state transitions.

To prevent repeated attempts and failures to bootstrap or restart the operating system, the console subsystem maintains two flags: the bootstrap-in-progress flag and the restart-in-progress flag. If a system bootstrap or restart would occur automatically but the corresponding flag is already set, the console assumes that an attempt has already been made and has failed. The console therefore does not try again.

To load and start (bootstrap) the operating system, the console searches for a section of correctly functioning system memory large enough to hold a primary bootstrap program (called VMB). If a section of memory is found, the console loads and starts VMB. VMB loads and starts the operating system.

The console uses the following algorithm to bootstrap the operating system:

1. If this bootstrap is the result of a console BOOT command, skip to step 4.
2. Print the message "Attempting system bootstrap" on the console terminal.
3. Check to see if the bootstrap-in-progress flag is set. If so, boot fails.
4. Set the bootstrap-in-progress flag.
5. Locate a page-aligned block of good memory. VAX Processors announced after 1987 must locate a block 256 kilobytes in length; earlier VAX processors locate a block 64 kilobytes in length.

   Recall that when memory management is disabled, virtual addresses are translated by the processor to a particular range of addresses in the physical address space through an implementation dependent mechanism. When locating the block of good memory, the processor must ensure that the entire block resides within this range of physical addresses. For example, consider an implementation with a 34-bit physical address space that translates a 32-bit virtual address to a 34-bit physical address by truncating the two high-order bits and sign

extending the remainder. This implementation must locate a block within the first 512 megabytes of physical memory.

If a block cannot be found which meets the above requirements, boot fails.

Note that in locating the block, the processor performs a test of memory which leaves the contents of memory UNPREDICTABLE.

6. Load a bootstrap program into that good memory, starting 512 bytes from the beginning. The name of the bootstrap program is VMB.EXE. If VMB cannot be found on the load device, or if there is an error during loading, boot fails.
7. Load the general registers:

   The contents of R0 through R4 are implementation dependent, but contain information describing:

R0	Boot device's type code.
R1	Boot device's bus address.
R2	Boot device's controller information.
R3	Boot device's unit number.
R4	Reserved.
R5	Boot control parameter from the boot command.
R6	Reserved.
R7	Reserved.
R8	Reserved.
R9	Reserved.
R10	The halt PC.
R11	The halt PSL.
AP	The halt code.
FP	Reserved.
SP	Address of 512 bytes past the start of good memory.

8. Start VMB at the address in SP. VMB loads and starts the operating system.

If bootstrap fails, the console prints a message reporting the failure. The message may explain the cause of the failure, or it may just report "System bootstrap failed."

If the bootstrap is successful, the operating system sends a message to the console, causing the console to clear the bootstrap-in-progress flag. See section 10.6 for a description of the messages the operating system can pass to the console.

10.2 System Restart

The console can restart a halted operating system. To do so, the console searches system memory for the restart parameter block (RPB), a data structure constructed for this purpose by the operating system. If a valid RPB is found, the console restarts the operating system at an address specified in the RPB.

The console keeps an internal flag, restart-in-progress, which it uses to avoid repeated attempts to restart a failing operating system. An additional restart-in-progress flag may be maintained by software in the RPB.

A system restart can occur as the result of a powerfail restart or as the result of a processor halt. See section 10.4 for a complete description.

The console uses the following algorithm to restart the operating system:

1. Print the message "Attempting system restart" on the console terminal.
2. Check to see if the internal restart-in-progress flag is set. If so, restart fails.
3. Set the internal restart-in-progress flag.
4. Check to see if memory has been preserved by battery back-up. If not, restart fails.
5. Look for an RPB left in memory by the operating system. If none is found, restart fails.
6. Read the software restart-in-progress flag from bit ⟨0⟩ of the fourth longword of the RPB. If it is set, restart fails.
7. Load SP with the address of the RPB plus 512.
8. Load AP with the halt code. Load R10 with the halt PC. Load R11 with the halt PSL.
9. Start the processor at the restart address, which is read from the second longword in the RPB.

If restart fails, the console prints a message reporting the failure. The message may explain the cause of the failure, or it may just report "System restart failed."

physical address of the RPB	:RPB
physical address of the restart routine	
checksum of the first 31 longwords of the restart routine	
software restart in progress flag (bit 0)	

Figure 10.1
Restart Parameter Block (RPB)

If the restart is successful, the operating system sends a message to the console, causing the console to clear its internal restart-in-progress flag. See section 10.6 for a description of the messages the operating system can pass to the console.

The RPB is a page-aligned control block created by the operating system, and is shown in Figure 10.1.

The console uses the following algorithm to find an RPB:

1. Search for a page of memory that contains its address in the first longword. If none is found, the search for an RPB has failed. Memory is searched linearly from low physical addresses to high physical addresses.
2. Read the second longword in the page (the physical address of the restart routine). If it is not a valid physical address, or if it is 0, return to step 1. The check for 0 is necessary to ensure that a page of zeros does not pass the test for a valid RPB.
3. Calculate the 32-bit two's complement sum (ignoring overflows) of the first 31 longwords of the restart routine. If the sum does not match the third longword of the RPB, return to step 1.
4. A valid RPB has been found.

Remember that when memory management is disabled, virtual addresses are translated by the processor to a particular range of addresses in the physical address space through an implementation dependent mechanism. When constructing the RPB, the operating system must ensure that both the entire RPB and the restart address specified within the RPB reside within this range of physical addresses.

10.3 System Powerfail and Recovery

The system requires power to operate. The system power supply conditions external power and transforms it for use by the processor. When external power fails, the power supply requests a powerfail interrupt of the processor. The power supply continues to provide power to the processor for at least 2 milliseconds after the interrupt is requested to allow the operating system to save state. When the power supply can no longer provide power to the processor, the processor is halted and powered off. Battery backup options are available on some processors to supply power after external power fails, to maintain the contents of main memory, and to keep system time with the time-of-day clock.

When power is restored, the console initializes itself, initializes the processor, and examines the front panel console-lock and auto-restart switches. If the console is locked, it attempts a system restart; if that fails, it attempts a system bootstrap. If the console is not locked, its action is determined by the setting of the auto-restart switch.

Note that when the processor loses power, its state is lost. For example, if a processor is halted when power fails, the action on power up is still determined by the front panel switches. So the system does not necessarily stay halted.

When power is restored, the processor initializes itself. There are three kinds of hardware initialization: processor initialization, system bus initialization, and power-up initialization. Processor initialization is the result of a console INITIALIZE and involves the initialization of registers internal to the processor and the console. System bus initialization is the result of a console UNJAM command and is implementation dependent. Power-up initialization affects the system as a whole. It is the result of the restoration of power and includes a processor initialization.

The processor must be initialized after an error halt. If the processor starts running after an error halt, without an intervening processor initialization, the operation of the processor is UNDEFINED.

The effects of processor initialization and power-up initialization

Table 10.1
Effects of Processor Initialization

Processor State	Value After Processor Initialization
PSL	041F0000 (hex)
IPL	1F (hex)
ASTLVL	4
SISR	0
ICCS	⟨6⟩ and ⟨0⟩ clear, the rest UNPREDICTABLE
RXCS	0
TXCS	80 (hex)
MAPEN	0
PME	0
ACCS	0 if no FPA; 8001 (hex) if FPA installed
cache, instruction buffer, or write buffer	empty or valid
console previous reference	physical address, longword size, address 0
KSP, ESP, SSP, USP, ISP	UNPREDICTABLE
P0BR, P0LR, P1BR, P1LR	UNPREDICTABLE
SBR, SLR	UNPREDICTABLE
PCBB	UNPREDICTABLE
SCBB	UNPREDICTABLE
translation buffer	UNPREDICTABLE
NICR, ICR	UNPREDICTABLE
TODR	unaffected
main memory	unaffected
registers R0 through PC	unaffected
halt code	unaffected
bootstrap-in-progress flag	unaffected
restart-in-progress flag	unaffected
vector registers V0 - V15	unaffected
VMR, VLR, VCR, VSAR	unaffected
VPSR, VAER	0

on the processor's state are summarized in Tables 10.1 and 10.2. Processor registers not listed in the tables have UNPREDICTABLE values after processor or power-up initialization.

10.4 Major System State Transitions

The transitions between major system states are determined by the current state and by a number of variables and events, including the following:

Table 10.2
Additional Effects of Power-Up Initialization

Processor State	Value After Power-Up Initialization
bootstrap-in-progress flag	cleared
internal restart-in-progress flag	cleared
halt code	03 (power-up)
general registers	UNPREDICTABLE
system memory	unaffected if preserved by battery back-up; otherwise, UNPREDICTABLE
TODR	unaffected if preserved by battery back-up; otherwise, 0
vector registers	UNPREDICTABLE
VMR, VLR, VCR, VSAR	UNPREDICTABLE

- Whether power is available to the system
- The console front panel auto-restart switch
- The console-lock switch
- The bootstrap-in-progress flag
- The restart-in-progress flag
- Processor error halts
- The HALT instruction
- Console commands

Table 10.3 shows the actions that cause major system state transitions.

The processor follows these rules:

- If the console is not locked when power is restored or when the processor halts, enter the state selected by the console front panel auto-restart switch.
- If the console is locked when power is restored or when the processor halts, attempt a system restart.
- When system restart fails, attempt a system bootstrap.
- When system bootstrap fails, halt.
- When system bootstrap or system restart succeeds, the processor starts running.
- When the processor is halted and the console is not locked, the console BOOT command causes a system bootstrap.

Table 10.3
Major System State Transitions

Initial State	Final State[a]				
	Powered Off	Halted	Booting	Restarting	Running
Powered Off		A and power restored	B and power restored	C and power restored	
Halted	powerfail		BOOT command and unlocked		START or CONTINUE and unlocked
Booting	powerfail	boot fails or D			boot succeeds
Restart	powerfail	D	restart fails		restart succeeds
Running	powerfail	A and processor halts, or D	B and processor halts	C and processor halts	

[a]Table entries describe the actions causing the transition. Capital letters represent the following actions and settings established by the console operator:
A: Console is unlocked and the halt action switch is set to Halt.
B: Console is unlocked and the halt action switch is set to Boot.
C: Console is unlocked and the halt action switch is set to restart, or the console is locked.
D: Console is unlocked and the operator types CTRL/P and HALT.

- When the processor is halted and the console is not locked, the operating system moving a value of 0F02 into the TXDB register causes a system bootstrap. (See section 10.6.1 for more information about TXDB.)
- When the processor is halted and the console is not locked, the console START and CONTINUE commands cause the processor to start running.
- If the console is not locked and is running, booting, or restarting, typing CTRL/P followed by a HALT command at the console halts the processor.

10.5 System Halted (Console I/O Mode)

Included in this section about the system-halted state are descriptions of the console; command syntax, keywords, language subsets; errors and error messages; and halt and halt messages.

10.5.1 Console

Traditionally, computers have had a panel of lights and switches on the front for processor diagnosis and for operation of stand-alone programs. On VAX, this function is provided by an ASCII console through which the operator controls the processor. The ASCII console may be envisioned as a virtual console processor attached to the main processor, to a console terminal, and to a console file-storage device. Note that the console processor need not be physically separate from the main processor. It may be implemented in main processor microcode, as in the VAX-11/750 computer system. The console processor interprets commands typed on the console terminal and controls the operation of the main processor.

Through the console terminal, an operator can boot the operating system, a field service engineer can maintain the system, and a system user can communicate with running programs. Sophisticated users may also use the console for developing software.

The processor can halt as the result of an operator command, a serious system error, a HALT instruction, or a powerfail recovery. (See section 10.4 for a complete description.) When the processor is halted, the operator controls the system through the console command language. The console is in console I/O mode. The console prompts the operator for input with the string of right angle brackets (>>>).

It may be possible for the operator to put the system in an inconsistent state through the use of the console commands. For example, it may be possible to use the console to set bits in MBZ fields or to set conflicting control bits. The operation of the processor in such a state is UNDEFINED.

10.5.2 Special Characters

In console I/O mode, several characters have special meanings. Some of these characters are produced by pressing a single key, while others, like the control characters, are produced by pressing the character while simultaneously pressing the control key (CTRL/).

- *Carriage return.* Typing a carriage return ends a command line. No action is taken on a command until after it is termi-

nated by a carriage return. A null line terminated by a carriage return is treated as a valid, null command. No action is taken, and the console again prompts for input. Carriage return is echoed as carriage return, line feed.

- *Rubout.* When the operator types rubout, the console deletes the character that the operator previously typed. The console echoes with a backslash (\), followed by the character being deleted. If the operator types additional rubouts, the additional characters deleted are echoed. When the operator types a character other than rubout, the console echoes another backslash, followed by the character typed. The result is to echo the characters deleted, surrounding them with backslashes. For example:

```
The operator types: EXAMI;E<rubout><rubout>NE<CR>
The console echoes: EXAMI;E\E;\NE<CR>
The console sees the command line: EXAMINE<CR>
```

  The console does not delete characters past the beginning of a command line. If the operator types more rubouts than there are characters on the line, the extra rubouts are ignored. If a rubout is typed on a blank line, it is ignored.
- CTRL/U – The console echoes ^U and deletes the entire line. If CTRL/U is typed on an empty line, it is echoed; otherwise, it is ignored. The console prompts for another command.
- CTRL/S – Typing CTRL/S stops console transmissions to the console terminal until CTRL/Q is typed. Additional input between CTRL/S and CTRL/Q is buffered as input but not echoed until CTRL/Q is typed. CTRL/S typed again before the CTRL/Q is ignored. CTRL/S and CTRL/Q are not echoed.
- CTRL/Q – Typing CTRL/Q resumes console transmissions stopped by CTRL/S. Additional typing of CTRL/Q is ignored. CTRL/S and CTRL/Q are not echoed.
- CTRL/O – Typing CTRL/O causes the console to throw away transmissions to the console terminal until the next CTRL/O is entered. CTRL/O is echoed as ^O⟨CR⟩ when it disables output; it is not echoed when it re-enables output. Output is re-enabled if the console prints an error message or if it prompts for a command from the terminal. Reading a command from a command file and displaying a REPEAT command do not re-enable output. When output is re-enabled for reading a command, the console prompt is displayed. Output is also enabled by entering program I/O mode by CTRL/P and by CTRL/C.

- CTRL/C – Typing CTRL/C causes the console to echo ^C and to abort processing of a command. CTRL/C has no effect as part of a binary load data stream. CTRL/C re-enables output stopped by CTRL/O. When CTRL/C is typed as part of a command line, the console deletes the line, as it does with CTRL/U.
- CTRL/P – If the console is in console I/O mode, CTRL/P is equivalent to CTRL/C and is echoed as ^P. If the console is in program I/O mode and is locked, CTRL/P is not echoed but is passed to the operating system like any other character. If the console is in program I/O mode and is not locked, CTRL/P is not echoed but causes the processor to enter console I/O mode. It is UNPREDICTABLE whether CTRL/P also causes the processor to halt. HALT subsequently must be typed to halt the processor.

If an unrecognized control character is typed (a control character here means a character with an ASCII code less than 32 decimal), it is echoed as caret followed by the character with ASCII code 64 greater. For example, BEL (ASCII code 7) is echoed as ^G, since capital G is ASCII code 7 + 64 = 71. When a control character is deleted with rubout, it is echoed the same way. After echoing the control character, the console processes it as a normal character. Unless the control character is part of a comment, the command will be invalid and the console will respond with an error message.

The response of the console to characters with codes greater than 127 (decimal) is UNPREDICTABLE.

10.5.3 *Command Syntax*

The console accepts commands of lengths up to 80 characters. Longer commands are responded to with an error message.

Commands may be abbreviated. Abbreviations are formed by dropping characters from the end of a keyword. All commands but SET may be unambiguously abbreviated to one character. SET cannot be abbreviated to less than two characters, since it then conflicts with START. The console verifies all characters typed in a command, even when they are not needed to uniquely identify the command.

Multiple adjacent spaces and tabs are treated as a single space by the console. Leading and trailing spaces and tabs are ignored.

Command qualifiers can appear after the command keyword or after any symbol or number in the command.

All numbers (addresses, data, counts) are in hexadecimal. (Note, though, that symbolic register names include decimal digits.) Hex digits are 0 through 9 and A through F. The console does not distinguish between uppercase and lowercase either in numbers or in commands. Both are accepted.

10.5.4 Commands

Following is a list of processor control, data transfer, and console control command keywords. These commands are described in the next section of this chapter.

Processor Control Commands

- INITIALIZE
- START ⟨address⟩
- CONTINUE
- HALT
- BOOT ⟨device⟩
- NEXT ⟨count⟩
- MICROSTEP ⟨count⟩
- UNJAM

Data Transfer Commands

- EXAMINE ⟨address⟩
- DEPOSIT ⟨address⟩ ⟨data⟩
- LOAD ⟨file⟩
- X ⟨address⟩ ⟨count⟩

Console Control Commands

- FIND
- REPEAT ⟨command⟩
- SET ⟨parameter⟩ ⟨value⟩
- TEST
- @ ⟨file⟩
- ! ⟨comment⟩

BOOT — Load and Start Primary Bootstrap

Purpose

The device specification is of the format "ddan," where "dd" is a two-letter device mnemonic, "a" is an optional alphabetic adapter identifier, and "n" is a one-digit unit number.

The console initializes the processor and starts VMB running. (See section 10.1.) VMB boots the operating system from the specified device. The default device is implementation dependent.

Format

```
BOOT [<qualifier list>] [<device>]
```

Qualifiers

/R5:⟨data⟩ After initializing the processor and before starting VMB, R5 is loaded with the specified data. This allows a command file containing a BOOT command or a console user to pass a parameter to VMB.

CONTINUE — Continue Execution

Purpose

The processor begins instruction execution at the address currently contained in the program counter. Processor initialization is not performed. The console enters program I/O mode.

The console may use memory bytes −8(ISP) through −1(ISP) in the process of starting the processor. The contents of these bytes are then UNPREDICTABLE.

Format

```
CONTINUE
```

DEPOSIT Deposit Data to Address

Purpose

This command deposits the data into the address specified. If no address space or data size qualifiers are specified, the defaults are the last address space and data size used in a DEPOSIT or EXAMINE command. After processor initialization, the default address space is physical memory, the default data size is long, and the default address is zero.

If the specified data value is too large to fit in the data size to be deposited, the console ignores the command and issues an error response. If the specified data value is smaller than the data size to be deposited, it is extended on the left with zeros.

Format

```
DEPOSIT [<qualifier list>] <address> <data>
```

Qualifiers

(If a qualifier that is invalid for a specified address is used, the console ignores the command and issues an error response.)

/B The data size is byte. Should not be used when accessing vector register elements.

/W The data size is word. Should not be used when accessing vector register elements, VCR, VLR, or VMR.

/L The data size is longword. When used with vector register elements: only the low-order longword of the specified elements are accessed; the qualifier has no effect on the address increment used by the /N qualifier and plus (+) and minus (–) symbolic addressing. Should not be used when accessing VCR, VLR, or VMR.

/Q (Optional.) The data size is quadword. For DEPOSIT, the data value is entered as a 64-bit hexadecimal number with no intervening spaces. For EXAMINE, the data value is displayed as two longwords separated by a single space. When accessing general registers, both R[n] and R[n+1] are accessed. For example:

```
D/Q R0 0123456789ABCDEF
```

loads R0 and R1. Using this qualifier when accessing

vector register elements or VMR is redundant but results in no error. This qualifier should not be used to access VCR, VLR, SP, or PC.

/FF (Optional.) The data value is entered and interpreted as an F_floating value. When used with vector register elements, only the low-order longword of the specified elements are interpreted, and the qualifier has no effect on the address increment used by the /N qualifier and plus (+) and minus (–) symbolic addressing. This qualifier should not be used when accessing VCR, VLR, or VMR. Note that not all F_floating point numbers can be rendered exactly in decimal representation; when the value of all bits must be known or set exactly, access the data as a hexadecimal longword by using the /L qualifier. An example is:

```
D/FF R0 3.6E-19
```

/DF (Optional.) The data value is entered and interpreted as a D_floating value. When accessing general registers, both R[n] and R[n+1] are accessed. This qualifier should not be used when accessing VCR, VLR, VMR, SP, or PC. Note that not all D_floating point numbers can be rendered exactly in decimal representation; when the value of all bits must be known or set exactly, access the data as a hexadecimal quadword (by using the /Q qualifier, for example). An example is:

```
D/DF R0 3.6E-19
```

/GF (Optional.) The data value is entered and interpreted as a G_floating value. When accessing general registers, both R[n] and R[n+1] are accessed. This qualifier should not be used when accessing VCR, VLR, VMR, SP, or PC. Note that not all G_floating point numbers can be rendered exactly in decimal representation; when the value of all bits must be known or set exactly, access the data as a hexadecimal quadword (by using the /Q qualifier, for example). An example is:

```
D/GF R0 3.6E301
```

/V The address space is virtual memory. All access and protection checking occur. If the access would not be allowed to a program running with the current PSL, the console issues an error message. This includes refusing odd address references if PSL⟨CM⟩ is set. Virtual space DEPOSITs cause the PTE⟨M⟩ bit to be set. If memory mapping is not enabled, virtual addresses are equal to physical addresses.

/P The address space is physical memory.

/I The address space is internal processor registers. These are the registers addressed by the MTPR and MFPR instructions.

/G The address space is the general register set, R0 through PC.

/M (Optional.) The address space is machine dependent.

/C The address space is microcode memory.

/U (Optional.) The address space is console microprocessor memory.

/N:⟨count⟩ The address is the first of a range. The console deposits to the first address, then to the specified number of succeeding addresses. The increment used to calculate the succeeding addresses is the data size in effect (either the default or the one specified by a qualifier on the command line). Even if the address is the symbolic address minus (–), the succeeding addresses are at larger addresses. The symbolic address specifies only the starting address, not the direction of succession. For repeated references to preceding addresses, use REPEAT DEPOSIT – ⟨data⟩.

When used to access vector registers (V0–V15), the number of elements and vector registers accessed is determined by the granularity of the symbolic address. If the symbolic address merely indicates a vector register with no element range specified, such as V0, then the console sets the address increment as the length of one vector register and accesses all the elements of successive vector registers. If the symbolic address indicates an element range, such as V0[1], then the console sets the address increment as one element and accesses the successive elements of

the specified vector register. In both cases, the increment used is unaffected by a data size qualifier that controls the size of the element values displayed or entered, such as /L or /FF.

Description

Here are some examples of the DEPOSIT command:

```
D/P/B/N:1FF 0 0
    Clears the first 512 bytes of physical memory.
D/V/L/N:3 1234 5
    Deposits 5 into four longwords in virtual memory.
D/N:8 R0 FFFFFFFF
    Loads general registers R0 through R8.
D/N:200 - 0
    Clears the previous address, then the next 512.
D/N:1 V0 FEDCBA9876543210
    Loads all elements of vector register V0 and V1
    with the same value.
D/N:1 V0[3E] FEDCBA9876543210
    Loads elements 3E and 3F of vector register V0
    with the same value.
D/L/N:1 V0[3E] 76543210
    Loads the low-order longword of elements 3E and 3F
    of vector register V0 with the same value.
```

Addresses

If conflicting address spaces are specified, the console ignores the command and issues an error response. The address may also be one of the following symbolic addresses:

PSL — Processor status longword. No address space qualifier is legal. When PSL is examined, the address space is set to M (machine dependent).

PC — Program counter (general register 15). The address space is set to /G.

SP — Stack pointer (general register 14). The address space is set to /G.

Rn — General register n. The register number is in decimal. The address space is set to /G. For example:

```
D R5 1234 is equivalent to D/G 5 1234
D R10 6FF00 is equivalent to D/G A 6FF00
```

VCR — Vector count register. No address space qualifier is legal; when examined, the address space is set to /VE.

VLR — Vector length register. No address space qualifier is legal; when examined, the address space is set to /VE.

VMR Vector mask register. No address space qualifier is legal; when examined, the address space is set to /VE.

Vn All elements of vector register n. The register number, n, is in decimal and is between 0 and 15. No address specifier is legal; when examined, the address space is set to /VE.

Vn[m] Element m of vector register n. The register number, n, is in decimal and is between 0 and 15. The element number, m, is in hexadecimal and is between 0 and 3F. No address specifier is legal; when examined, the address space is set to /VE.

+ Plus. The location immediately following the last location referenced in an EXAMINE or a DEPOSIT. For references to physical or virtual memory spaces, the location referenced is the last address plus the size of the last reference (1 for byte; 2 for word; 4 for long and F_floating; 8 for quad, D_floating, and G_floating). For vector registers, the granularity of the symbolic address (either Vn or Vn[m]) determines whether the next element or the next vector register is accessed; this is unaffected by a data size qualifier such as /L. (See discussion under /N qualifier for more detail.) For other address spaces, the address is the last address referenced, plus one.

– Minus. The location immediately preceding the last location referenced in an EXAMINE or a DEPOSIT. For references to physical or virtual memory spaces, the location referenced is the last address minus the size of this reference (1 for byte; 2 for word; 4 for long and F_floating; 8 for quad, D_floating, and G_floating). For vector registers, the granularity of the symbolic address (either Vn or Vn[m]) determines whether the preceding element or the preceding vector register is accessed; this is unaffected by a data size qualifier such as /L. (See discussion under /N qualifier for more detail.) For other address spaces, the address is the last address referenced minus one.

* Asterisk. The location last referenced in an EXAMINE or a DEPOSIT.

@ At sign. The location addressed by the last location referenced in an EXAMINE or DEPOSIT. This symbolic address should not be used when the last location referenced is a vector register or vector register element.

Data Format

If conflicting data sizes are specified, the console ignores the command and issues an error response.

When depositing a data value to a vector register element or the VMR, the default for data size and format is a quadword (regardless of whether the /Q qualifier is supported); thus, the value is entered on the command line as a 64-bit hexadecimal number with no intervening spaces between longwords:

```
D V1[3F] 0123456789ABCDEF
D VMR AAAAAAAAAAAAAAAA
```

When accessing VCR or VLR, the default data size is a byte.

EXAMINE Examine Data at Address

Purpose

This command examines the contents of the specified address. If no address is specified, the plus sign (+) is assumed. The same qualifiers may be used on EXAMINE as may be used on DEPOSIT. The address may also be one of the symbolic addresses described under DEPOSIT.

Format

```
EXAMINE [<qualifier list>] [<address>]
```

Response

The processor responds as follows:

```
<tab><address space identifier> <address> <data>
```

The address space identifier can be:

- P Physical memory. Note that when virtual memory is examined, the address space and address in the response are the translated physical address.
- G General register.
- I Internal processor register.
- M Machine-dependent address space. When the PSL is examined, the address space identified is machine dependent.
- C Microcode memory.
- U (Optional) Console microprocessor memory.
- VE Vector Processor data: used for vector register elements, VCR, VLR, and VMR.

Displaying Data

Quadword data is displayed as two longwords separated by a single space:

```
let:      R0 = 89ABCDEF  and   R1 = 01234567
then:     E/Q R0
returns:  G  R0  01234567 89ABCDEF
```

Multiple vector register elements are displayed in pairs as

```
VE  Vn[m]  <data>   Vn[m+1]  <data>
```

An entire vector register is displayed as

```
VE  Vn[00]  <data>   Vn[01]  <data>
    .
    .
    .
VE  Vn[3E]  <data>   Vn[3F]  <data>
```

VCR, VLR, and VMR are displayed as in this example:

```
VE  VLR  <data>
```

FIND — Find Good Memory for Bootstrap Program

Purpose

The console searches main memory starting at address zero for a page-aligned, 64 or 256-kilobyte block of good memory, or a restart parameter block (RPB). If the block is found, its address plus 512 is left in SP. If the block is not found, an error message is issued, and the contents of SP are UNPREDICTABLE. If no qualifier is specified, /RPB is assumed.

Format

```
FIND [<qualifier list>]
```

Qualifiers

/MEMORY — Search memory for a page-aligned block of good memory. VAX processors announced after 1987 are required to search for a block 256 kilobytes long; earlier VAX processors search for a block 64 kilobytes long. Since the search may include a read and write test of memory, the search leaves the contents of memory UNPREDICTABLE.

/RPB — Search memory for a restart parameter block. See section 10.2 for the search algorithm. The search leaves the contents of memory unchanged.

HALT — Halt Processor

Purpose

The processor stops execution of macroinstructions after completing the current macroinstruction. Neither processor initialization nor I/O initialization occurs, so I/O operations already in progress are unaffected. If the processor is already halted, the HALT command has no effect.

Format

```
HALT
```

Response

The processor responds as follows:

```
PC = <PC>
```

If the processor is already halted, the response is preceded by the halt message "Already halted."

INITIALIZE Initialize Processor

Purpose

A processor initialization is performed. See section 10.3 for initial register contents.

Format

```
INITIALIZE
```

LOAD Load File into Memory

Purpose

The console loads the contents of the specified file into memory. If no qualifiers are specified, the file is loaded into physical memory starting at address 0. If an unrecoverable device or memory error occurs during the load, the command is aborted and the console issues an error message.

Format

```
LOAD [<qualifier list>] <file>
```

Qualifiers

/S:⟨address⟩	The file is loaded starting at the specified address.
/C	The file is to be loaded into microcode memory.
/U	(Optional) The file is to be loaded into console microprocessor memory.

MICROSTEP Execute Microinstruction

Purpose

The console causes the processor to execute the specified number of microinstructions. If no count is specified, 1 is assumed. After the last microinstruction is executed, the console enters space-bar-step mode.

In space-bar-step mode, the console executes one microinstruction each time the operator presses the space bar. If the operator presses any other key, the console exits space-bar-step mode, then processes the character typed. Typing a carriage return is the suggested means of exiting from space-bar-step mode.

The operator can use the NEXT command to cause the console to finish the macroinstruction executing.

Format

```
MICROSTEP [<count>]
```

Response

The processor responds as follows:

```
uPC = <uPC>
```

NEXT Execute Next VAX Instruction

Purpose

The console causes the processor to execute the specified number of macroinstructions (VAX instructions). If no count is specified, 1 is assumed. After the last macroinstruction is executed, the console enters space-bar-step mode.

In space-bar-step mode, the console executes one macroinstruction each time the operator presses the space bar. If the operator presses any other key, the console exits space-bar-step mode, then processes the character typed. Typing a carriage return is the suggested means of exiting from space-bar-step mode.

The NEXT command finishes a macroinstruction partially executed by MICROSTEP. This partial execution is counted by NEXT as though it were the execution of a full instruction.

Format

```
NEXT [<count>]
```

Response

The processor responds as follows:

```
PC = <PC>
```

REPEAT — Repeat Console Command

Purpose

The console repeatedly displays and executes the specified command. The repeating is stopped when the operator types CTRL/C. Any valid console command may be specified for this command, except the REPEAT command and the @ command. If the command is REPEAT or @, the results are UNPREDICTABLE.

Format

```
REPEAT <command>
```

Response

(dependent on command specified)

SET — Set Console Parameter

Purpose

Sets the console parameter to the indicated value. The console parameters and their meanings are all implementation dependent.

Format

```
SET <parameter> <data>
```

SHOW — Show Console Parameter

Purpose

Shows the value of the indicated console parameter. The console parameters and their meanings are all implementation dependent.

Format

```
SHOW <parameter>
```

START — Start Execution

Purpose

The console starts instruction execution at the specified address. The default address is implementation dependent. If no qualifier is present, macroinstruction execution is started. If memory mapping is enabled, macroinstructions are executed from virtual memory. The START command is equivalent to a DEPOSIT to PC followed by a CONTINUE. No INITIALIZE is performed.

The console may use memory bytes –8(ISP) through –1(ISP) in the process of starting the processor. The contents of these bytes are then UNPREDICTABLE.

Format

```
START [<qualifier list>] [<address>]
```

Qualifiers

/C	Microinstruction (rather than macro) execution is started.
/U	(Optional) Console microprocessor instruction execution is started.

TEST — Perform Self-Test

Purpose

The console executes a self-test. All qualifiers are optional.

Format

```
TEST [<qualifier list>]
```

UNJAM — Initialize System Bus

Purpose

A system bus initialization is performed. The effects of a system bus initialization are implementation dependent.

Format

```
UNJAM
```

X Binary Load and Unload

Purpose

The X command is for use by automatic systems communicating with the console. It is not intended for use by operators. The console loads or unloads (that is, writes to memory or reads from memory) the specified number of data bytes, starting at the specified address. If no qualifiers specify otherwise, data is transferred to or from physical memory.

If bit ⟨31⟩ of the count is clear, data is to be received by the console and deposited into memory. If bit ⟨31⟩ of the count is set, data is to be read from memory and sent by the console. The remaining bits in the count are a positive number indicating the number of bytes to load or unload.

The console accepts the command upon receiving the carriage return. The next byte the console receives is the command checksum, which is not echoed. The command checksum is verified by adding all command characters, including the checksum (but not including the terminating carriage return, rubouts, or characters deleted by rubout), into an 8-bit register initially set to zero. If no errors occur, the result is zero. If the command checksum is correct, the console responds with the input prompt and either sends data to the requester or prepares to receive data. If the command checksum is in error, the console responds with an error message. The intent is to prevent inadvertent operator entry into a mode where the console is accepting characters from the keyboard as data with no escape sequence possible.

If the command is a load (bit ⟨31⟩ of the count is clear), the console responds with the input prompt, then accepts the specified number of bytes of data for depositing to memory and an additional byte of received data checksum. The data is verified by adding all data characters and the checksum character into an 8-bit register initially set to zero. If the final contents of the register are non-zero, the data or the checksum is in error, and the console responds with an error message.

If the command is a binary unload (bit ⟨31⟩ of the count is set), the console responds with the input prompt followed by the specified number of bytes of binary data. As each byte is sent, it is added

to a checksum register initially set to zero. At the end of the transmission, the two's complement of the low byte of the register is sent.

If the data checksum is incorrect on a load, or if memory errors or line errors occur during the transmission of data, the entire transmission is completed and then the console issues an error message. If an error occurs during loading, the contents of the memory being loaded are UNPREDICTABLE.

If the console implements the SET TERMINAL ECHO and SET TERMINAL NOECHO commands, the state of the echo flag is unaffected by the X command. Regardless of the flag, echo is suppressed while data strings and checksums are being received.

It is possible to control the console through the use of the console control characters (CTRL/C, CTRL/S, CTRL/O, for example) during a binary unload. It is not possible during a binary load because all received characters are valid binary data.

Data being loaded with a binary load command must be received by the console at a rate of at least one byte per second. If the console does not receive a data byte for more than one second, the console aborts the transmission by issuing an error message and prompting for input.

The entire command, including the checksum, may be sent to the console as a single burst of characters at the console's specified character rate. To make this command useful in automated systems, the console is able to receive at least 4K bytes of data in a single X command.

Format

```
X [<qualifier list>] <address> <count> <CR> <checksum>
```

Qualifiers

/P	Data is read from or written to physical memory.
/C	Data is read from or written to microcode memory.
/U	(Optional) Data is read from or written to console microprocessor memory.

@ The Indirect Command

Purpose

The console reads and executes commands from the specified file. The commands are displayed on the console terminal as they are read. When a BOOT, START, or CONTINUE command is executed, putting the console into program I/O mode, command file processing is suspended. If a "software done" message is received by the console (see section 10.6) and the processor halts, command file processing is continued. If the processor halts before a "software done" message is received by the console, the remainder of the command file is ignored.

Command files can be chained by using another @ command as the last command in a file. If an @ command is encountered in the middle of a command file, the console executes it but may ignore the remainder of the original command file. It is an implementation option whether the console resumes execution of the original command file on completion of the secondary file.

Format

```
@ <file>
```

! The Comment Line

Purpose

The comment is ignored.

Format

```
! <comment>
```

10.5.5 Command Language Subsets

To reduce cost, some implementations may not implement the full console command set. A subset implementation is defined.

The commands supported by a subset console are as follows:

- BOOT ⟨device⟩
- CONTINUE
- DEPOSIT ⟨address⟩ ⟨data⟩
- EXAMINE [⟨address⟩]
- INITIALIZE
- HALT
- START ⟨address⟩
- TEST
- X ⟨address⟩ ⟨count⟩
- ! ⟨comment⟩

EXAMINE and DEPOSIT support the qualifiers /B, /W, /L, /P, /V, /I, /G, and the symbolic address PSL.

The control characters supported are carriage return, CTRL/P, CTRL/S, CTRL/Q, CTRL/U, and rubout.

The subset console may perform range checking on addresses and data. If it does not, it truncates values that are too large and uses the lower digits.

The subset console may accept only abbreviated commands. It may also limit the command length to less than 80 characters. It may accept only uppercase commands. Automatic systems communicating with a console must limit themselves to the commands in the subset, must abbreviate all commands, and must use only uppercase if they are to communicate with any console implementation.

10.5.6 Vector Support

Support for console access of vector registers, VCR, VLR, and VMR is required by a VAX system that implements the VAX vector processor. This support requires all vector-related features as described in section 10.5.4 with the exception of the /Q, /FF, /DF, and /GF qualifiers for the EXAMINE and DEPOSIT commands.

10.5.7 Options

Some features are optional, such as the diagnosis mode and the /M and /U qualifiers. The /Q, /FF, /DF, and /GF qualifiers are also optional and may be implemented separately. These features may be implemented by any console, even by a subset console.

10.5.8 Errors and Error Messages

The console can issue error messages in response to commands. The case (uppercase or lowercase) is implementation dependent.

The console responds to all commands within 1 second. If the processor does not respond to a console request, the console issues an error message within 1 second.

The following three messages indicate failure of the requested operation. Some implementations may abbreviate some or all of these messages to "Can't."

- *"Can't power up."* The console microprocessor cannot complete its own power-up initialization. The state of the console and that of the processor is UNDEFINED.
- *"File not found."* The file specified in a BOOT, LOAD, or @ command cannot be found.
- *"Reference not allowed."* The requested reference is not allowed because of one of the following: the reference would violate virtual memory protection, the address is not mapped, the reference is invalid in the specified address space, or the value is invalid in the specified destination.

The following messages are responses to ill-formed commands. Some implementations may abbreviate some or all of these messages to "Illegal command."

- *"Illegal command."* The command string cannot be parsed.
- *"Invalid digit."* A number has an invalid digit.
- *"Line too long."* The command was too large for the console to buffer. The message is issued only after receipt of the terminating carriage return.
- *"Illegal address."* The address specified falls outside the limits of the address space.
- *"Value too big."* The value specified does not fit in the destination.

- *"Conflicting switches."* For example, two different data sizes are specified with an EXAMINE command.
- *"Unknown switch."* The switch is unrecognized.
- *"Unknown symbol."* The symbolic address in an EXAMINE or DEPOSIT is unrecognized.

The following message is produced when a binary transfer command is improperly specified.

- *"Incorrect checksum."* The command or data checksum of an X command is incorrect. If the data checksum is incorrect, this message is issued and is not abbreviated to "Illegal command."

The following message is produced when a HALT command is given to the console and the processor is already halted.

- *"Already halted."* The operator entered a HALT command and the processor was already halted.

Some console commands may result in errors. For example, if a memory error occurs as the result of a console command, the console will respond with an error message. Such errors do not affect the halted program. Specifically, the processor stays halted, and if it is started later, no exception or interrupt occurs as the result of the console error.

10.5.9 Halts and Halt Messages

Whenever the processor halts, the console prints the response "PC = ⟨PC⟩," where ⟨PC⟩ is the actual PC value at which the processor halted. Except when the halt was requested by a console HALT command or a NEXT command, the response is preceded by a halt message. For example:

```
?06 HALT executed
    PC = 800050D3
```

The number preceding the halt message is the halt code and is passed to the operating system on a restart. Halt code 03 does not have a corresponding message. It is passed by the console during powerfail restart.

The halt messages are as follows:

- *"!00 CPU halted."* The operator entered a HALT command while the processor was running, so the console halted the processor.
- *"!01 Microverify complete."* The console quick-verify completed successfully.
- *"!02 CPU halted."* The operator typed CTRL/P while the console was in program I/O mode. The console was not locked, and the console halted the processor.
- *03.* Halt code 03 does not appear in a halt message but is passed by the console on powerfail restart.
- *"!04 I-stack not valid."* In attempting to push state on the interrupt stack during an interrupt or exception, the processor discovered that the interrupt stack was mapped NO ACCESS or NOT VALID.
- *"!05 CPU double error."* The processor attempted to report a machine check to the operating system, and a second machine check occurred.
- *"!06 HALT executed."* The processor executed a HALT instruction in kernel mode.
- *"!07 Invalid SCB vector."* The vector had bits ⟨1:0⟩ set.
- *"!08 No user WCS."* An SCB vector had bits ⟨1:0⟩ equal to 2, and no user-writable control store was installed.
- *"!09 Error pending on halt."* The processor was halted (with a CTRL/P) before it could perform an error halt.
- *"!0A CHM from I-stack."* A change mode instruction was executed when PSL⟨IS⟩ was set.
- *"!0B CHM to interrupt stack."* The exception vector for a change mode had bit ⟨0⟩ set.
- *"!0C SCB read error."* A hard memory error occurred while the processor was trying to read an exception or interrupt vector.

10.6 System Running (Program I/O Mode)

When the processor is running, the console is in program I/O mode. In this mode, all terminal interaction is handled by the operating system. The console terminal becomes like any other operating system terminal and passes through all characters (except for CTRL/P). If the console is locked, even CTRL/P is passed through. If the console is not locked, CTRL/P causes the processor to halt and the console to enter console I/O mode.

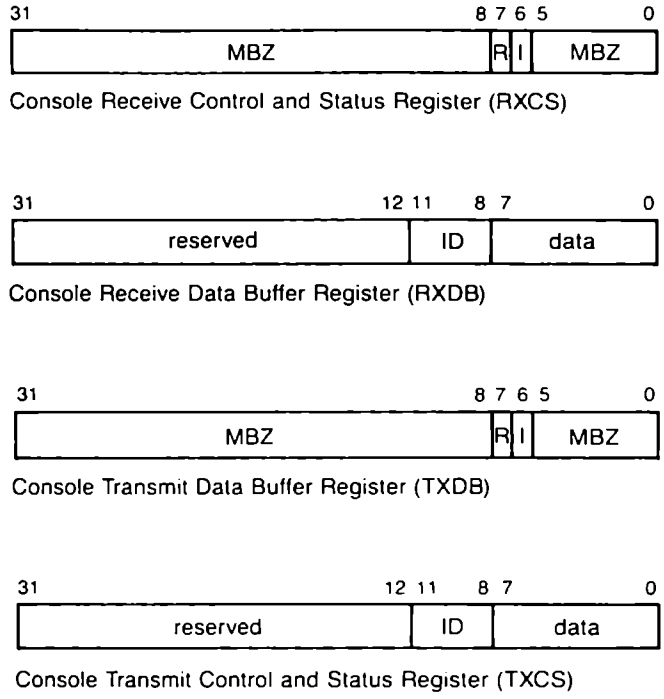


Figure 10.2
Console Terminal Registers

10.6.1 Console Terminal Registers

The console is accessed by the operating system through four internal processor registers. Two registers are associated with passing information from the console to the processor (receive registers) and two with passing information from the processor to the console (transmit registers). In each direction, there is a control and status register and a data buffer register. The registers are shown in Figure 10.2. The fields of the registers are described in Table 10.4.

Table 10.4
Fields of the Console Terminal Registers

Name	Extent	Description
Fields of the RXCS register		
RDY	⟨7⟩	Read Only. Cleared by processor initialization and by reading RXDB. When RXCS⟨RDY⟩ is clear, RXDB is UNPREDICTABLE. When RXCS⟨RDY⟩ is set, RXDB contains valid data and is ready to be read.
IE	⟨6⟩	Read/write. Cleared by processor initialization and by being written zero. If RXCS⟨IE⟩ is set by software while RXCS⟨RDY⟩ is already set, or if RXCS⟨RDY⟩ is set by the console while RXCS⟨IE⟩ is already set, then an interrupt is requested at IPL 14 (hex). That is, an interrupt is requested whenever the function RXCS⟨IE⟩ AND RXCS⟨RDY⟩ changes from 0 to 1.
Fields of the RXDB register		
ERR	⟨15⟩	An error occurred while receiving data, such as data overrun or loss of carrier. Cleared by processor initialization and by reading from RXDB.
ID	⟨11:8⟩	If zero, then data is identified as being from the console terminal. If nonzero, then the rest of the register is implementation dependent. Cleared by processor initialization and by reading from RXDB.
data	⟨7:0⟩	Data from the console terminal (if RXDB⟨ID⟩ is zero). UNPREDICTABLE unless RXCS⟨RDY⟩ is set.
Fields of the TXCS register		
RDY	⟨7⟩	Read only. Set by processor initialization. TXCS⟨RDY⟩ is clear when the console terminal is busy writing a character written to TXDB. TXCS⟨RDY⟩ is set when the console terminal is ready to receive another character.
IE	⟨6⟩	Read/write. Cleared by processor initialization and by being written clear. If TXCS⟨IE⟩ is set when TXCS⟨RDY⟩ becomes set, or if TXCS⟨IE⟩ set by software when TXCS⟨RDY⟩ is already set, an interrupt is requested at IPL 14 (hex). That is, an interrupt is requested whenever the function TXCS⟨IE⟩ AND TXCS⟨RDY⟩ changes from 0 to 1.
Fields of the TXDB register		
ID	⟨11:8⟩	If TXDB⟨ID⟩ is written zero when TXDB is written, the data goes to the console terminal. If TXDB⟨ID⟩ is written with 0F (hex), the data is a message to be sent to the console. If TXDB⟨ID⟩ is neither zero nor 0F (hex), the meaning is implementation dependent.
data	⟨7:0⟩	If TXDB⟨ID⟩ is zero, the data is a character sent to the console terminal to type. If TXDB⟨ID⟩ is 0F (hex), the data is a message to be sent to the console, with one of the following meanings: • *Software done.* A program started by a console indirect command file is signaling successful completion. When the processor halts, the console should resume processing the indirect command file. • *Boot processor.* The console should initiate a system bootstrap. • *Clear restart-in-progress flag.* A system restart has successfully completed. If a system restart would occur automatically, the attempt should be allowed. • *Clear bootstrap-in-progress flag.* A system bootstrap has successfully completed. If a system bootstrap would occur automatically, the attempt should be allowed.

11

Architectural Subsetting

This chapter describes those parts of the VAX architecture that may be included as standard features of a processor, provided as options to a processor, or omitted completely from a processor.

The subsetting of the architecture reflects the need to be able to trade off manufacturing cost, software development cost, and performance of VAX processors. The following conflicting hardware and software goals influenced the design of these rules:

- Hardware goal – Permit an implementor of a low-end processor to omit instructions and other features to reduce manufacturing cost without losing the ability to run all of the system software. The decision not to implement some options may impact the performance of certain software products.
- Software goal – Provide as small a number of classes of processor instruction sets as possible to reduce software development costs. In particular, a single version of each compiler or other layered software product should run on all processors in the VAX family. Also, the combination of hardware and instruction emulation routines in operating systems must (as required) give the appearance of a complete architecture on all processors.

11.1 Instruction-Set Options

There are four categories for instructions that define how the native, VAX instruction set may be implemented. After 1986, VAX processors must implement instructions in the base instruction

Table 11.1
Implementation of Instruction Groups by VAX Systems

VAX System	Base[a]	Packed Decimal	Extended Accuracy	Emulated FP	Emulated String	Vector	PDP-11 Mode
11/780, 11/785	y[b]	y	opt	y[c]	y	n	y
11/750	y[b]	y	opt	y[c]	y	n	y
11/725, 11/730	y	y	y	y	y	n	y
MicroVAX I/D	y[d]	n	n	y[c]	n	n	y
MicroVAX I/G	y[d]	n	n	y[c]	n	n	y
82xx, 83xx, VS8000	y	y	y	y	y	n	n
86xx	y	y	y	y	y	n	y
MicroVAX II	y[e]	n	n	y[c]	n	n	n
85xx, 87xx, 88xx	y	y	y	y	y	n	n
6000-2xx, 6000-3xx, 3xxx	y	n	n	y[c]	n	n	n
6000-4xx	y	n	n	n	n	opt	n
4000-3xx	y	n	n	n	n	n	n
9000	y	y	y	n	n	opt	n

[a]The letter "y" indicates instructions implemented in hardware/microcode; "n" instructions emulated by operating system software; "opt" instructions implemented in hardware/microcode as an option.
[b]G_floating-point instructions belonging to the base instruction group were implemented in hardware/microcode as an option.
[c]ACBx, EMODx, POLYx were implemented in hardware/microcode only for those floating-point datatypes that the system supported by hardware/microcode.
[d]MicroVAX I systems were available with either D_floating-point or G_floating-point instructions, but not both. 90% of all MicroVAX I systems sold implemented the D_floating-point instruction set.
[e]CMPC3, CMPC5, LOCC, SKPC, SCANC, SPANC were emulated by the MicroVAX II system.

group. Instructions in either an application extension group or the emulated-only instruction group may not be implemented, using the rules stated in this section. Vector instructions form a new category, and must be implemented or omitted as a whole. In addition, to the four native instruction categories, some VAX processors also support the execution of PDP-11 instructions through a compatibility mode.

Table 11.1 shows how the instruction-set options are implemented by various VAX systems.

Table 11.2
Base Instruction Group (Part 1)

Integer arithmetic and logical							
ADAWI	ADDB2	ADDB3	ADDL2	ADDL3	ADDB2	ADDB3	ADWC
ASHL	ASHQ	BICB2	BICB3	BICL2	BICL3	BICW2	BICW3
BISB2	BISB3	BISL2	BISL3	BISW2	BISW3	BITB	BITL
BITW	CLRB	CLRL	CLRQ	CLRW	CMPB	CMPL	CMPW
CVTBL	CVTBW	CVTLB	CVTLW	CVTWB	CVTWL	DECB	DECL
DECW	DIVB2	DIVB3	DIVL2	DIVL3	DIVW2	DIVW3	EDIV
EMUL	INCB	INCL	INCW	MCOMB	MCOML	MCOMW	MNEGB
MNEGL	MNEGW	MOVB	MOVL	MOVQ	MOVW	MOVZBW	MOVZBL
MOVZWL	MULB2	MULB3	MULL2	MULL3	MULW2	MULW3	PUSHL
ROTL	SBWC	SUBB2	SUBB3	SUBL2	SUBL3	SUBW2	SUBW3
TSTB	TSTL	TSTW	XORB2	XORB3	XORL2	XORL3	XORW2
XORW3							
Address							
MOVAB	MOVAL	MOVAQ	MOVAW	PUSHAB	PUSHAL	PUSHAQ	PUSHAW
Variable-length bit field							
CMPV	CMPZV	EXTV	EXTZV	FFC	FFS	INSV	
Branch and control							
ACBB	ACBL	ACBW	AOBLEQ	AOBLSS	BBC	BBCC	BBCCI
BBCS	BBS	BBSC	BBSS	BBSSI	BEQ	BGEQ	BGEQU
BGTR	BGTRU	BLBS	BLBC	BLEQ	BLEQU	BLSS	BLSSU
BNEQ	BRB	BRW	BSBB	BSBW	BVC	BVS	CASEB
CASEL	CASEW	JMP	JSB	RSB	SOBGEQ	SOBGTR	
Procedure call							
CALLG	CALLS	RET					

11.1.1 *Base Instruction Group*

The base instruction group is defined by exception; it is those instructions that may not be omitted in VAX processors announced after 1986. Of the 304 native mode VAX instructions, 242 of those instructions belong to the base instruction group. Wherever possible, VAX processors will optimize the performance of instructions in this group. VAX base system software (operating systems, runtime libraries, utilities, and code generated by Digital compilers) will target instruction use to this group. Instructions in the base group are shown in Table 11.2.

Table 11.2
Base Instruction Group (Part 2)

Miscellaneous							
BICPSW	BISPSW	BPT	INDEX	MOVPSL	NOP	POPR	PUSHR
HALT							
Queue							
INSQHI	INSQTI	INSQUE	REMQHI	REMQTI	REMQUE		
Operating system							
CHMK	CHME	CHMS	CHMU	HALT	LDPCTX	MFPR	MTPR
PROBER	PROBEW	REI	SVPCTX				
Character string							
CMPC3	CMPC5	LOCC	MOVC3	MOVC5	SCANC	SKPC	SPANC
F_floating point							
ADDF2	ADDF3	CMPF	CVTBF	CVTFB	CVTFL	CVTFW	CVTLF
CVTRFL	CVTWF	DIVF2	DIVF3	MNEGF	MOVF	MULF2	MULF3
SUBF2	SUBF3	TSTF					
D_floating point							
ADDD2	ADDD3	CMPD	CVTBD	CVTDB	CVTDF	CVTDL	CVTDW
CVTFD	CVTLD	CVTRDL	CVTWD	DIVD2	DIVD3	MNEGD	MOVD
MULD2	MULD3	SUBD2	SUBD3				
TSTD							
G_floating point							
ADDG2	ADDG3	CMPG	CVTBG	CVTFG	CVTGB	CVTGF	CVTGL
CVTGW	CVTLG	CVTRGL	CVTWG	DIVG2	DIVG3	MNEGG	MOVG
MULG2	MULG3	SUBG2	SUBG3				
TSTG							

11.1.2 *Application-Extension Instruction Groups*

Instructions in an application extension group may be implemented or omitted only as a group. That is, if any instruction in one of the application groups is implemented (omitted), all instructions in that application extension group are implemented (omitted). All VAX processors announced after 1986 have the option to implement zero or more of the application extension groups. Emulation software is required for instructions in unsupported application groups. VAX compilers will support features to complement the application extension groups implemented in VAX processors. There are two application extension groups; these are shown in Table 11.3.

Table 11.3
Application-Extension Instruction Groups

Decimal							
ADDP4	ADDP6	ASHP	CMPP3	CMPP4	CVTLP	CVTPL	CVTPS
CVTPT	CVTSP	CVTTP	DIVP	MOVP	MULP	SUBP4	SUBP6
Extended-accuracy							
ADDH2	ADDH3	CLRH	CMPH	CVTBH	CVTDH	CVTFH	CVTGH
CVTHB	CVTHD	CVTHF	CVTHG	CVTHL	CVTHW	CVTLH	CVTRHL
CVTWH	DIVH2	DIVH3	MNEGH	MOVAH	MOVH	MOVO	MULH2
MULH3	PUSHAH	SUBH2	SUBH3	TSTH			

- A packed-decimal-string group with sixteen instructions. If an instruction in this group is omitted by a processor, execution of the instruction results in an instruction-emulation exception.
- An extended-accuracy group with twenty-nine H_floating and octaword instructions. If an instruction in this group is omitted by a processor, execution of the instruction results in a reserved-instruction fault.

11.1.3 *Emulated-Only Instruction Group*

No VAX processor announced after 1986 is required to implement an instruction in the emulated-only instruction group. VAX processors may implement some or all of these instructions, but are not required to implement any. Operating system software supports emulation of instructions in this group. Instructions in the emulated-only instruction group, shown in Table 11.4, are classified into one of two groups.

- Emulated string instructions. If an instruction in this group is omitted by a processor, execution of the instruction results in an instruction-emulation exception.
- Emulated floating-point instructions. If an instruction in this group is omitted by a processor, execution of the instruction results in a reserved-instruction fault.

11.1.4 *Vector Processor and Vector Instructions*

A processor may implement or omit the vector processor only as a whole. That is, if any vector facility (instruction or register) is implemented (omitted), all vector facilities are implemented (omitted). If a processor omits the vector processor, execution of

Table 11.4
Emulated-Only Instruction Group

Emulated floating point							
ACBF	ACBD	ACBG	ACBH	EMODF	EMODD	EMODG	EMODH
POLYF	POLYD	POLYG	POLYH				
Emulated string							
CRC	EDITPC	MATCHC	MOVTC	MOVTUC			

Table 11.5
Vector Instruction Group

IOTA	MFVP	MTVP	VGATHL	VGATHQ	VLDL	VLDQ	VSADDD
VSADDF	VSADDG	VSADDL	VSBICL	VSBISL	VSCATL	VSCATQ	VSCMPD
VSCMPF	VSCMPG	VSCMPL	VSDIVD	VSDIVF	VSDIVG	VSMERGE	VSMULD
VSMULF	VSMULG	VSMULL	VSSLLL	VSSRLL	VSSUBD	VSSUBF	VSSUBG
VSSUBL	VSTL	VSTQ	VSXORL	VSYNC	VVADDD	VVADDF	VVADDG
VVADDL	VVBICL	VVBISL	VVCMPD	VVCMPF	VVCMPG	VVCMPL	VVCVT
VVDIVD	VVDIVF	VVDIVG	VVMERGE	VVMULD	VVMULF	VVMULG	VVMULL
VVSLLL	VVSRLL	VVSUBD	VSSUBF	VSSUBG	VSSUBL	VVXORL	

a vector instruction results in a reserved-instruction fault. Vector instructions are shown in Table 11.5.

11.1.5 PDP-11 Compatibility Mode

No VAX processor announced after 1983 is required to implement PDP-11 compatibility mode. In a processor without compatibility mode, the execution of an REI instruction attempting to enter compatibility mode results in a reserved-operand fault.

11.2 Other Implementation Options

Certain other features of the VAX architecture are optional in VAX processor implementations.

11.2.1 Internal Processor Registers

The following registers are optional for VAX processors that were announced after 1983. If any of the registers named on one of the following lines is included, all the registers on that line must be included.

- Interval timer registers: NICR, ICR, ICCS except for ⟨IE⟩. (That is, implementation of ICCS⟨IE⟩ is required.)
- Time-of-Year clock register: TODR
- Console registers: RXCS, RXDB, TXCS, TXDB
- Performance Monitor Enable register: PME

11.2.2 *Memory Management*

The VAX memory management includes a PTE format with a 21-bit page-frame number (PFN), a hardware-maintained modify bit, and physical addresses in the SCBB, SBR, and PCBB registers. VAX processors announced after 1987 may support a PTE format with a 25-bit PFN, a software-maintained modify bit, and virtual addresses in the SCBB and PCBB registers. See Chapter 4 for more detail.

11.2.3 *rtVAX Memory Management*

The rtVAX is a variant of the VAX Architecture. An rtVAX processor implements the per-process page tables in physical memory. All other VAX processors implement the per-process page tables in virtual memory. Therefore, translation of process-space addresses is different on an rtVAX than as described in Chapter 4.

On an rtVAX processor, the process-space page tables are located in physical memory, and the P0BR and P1BR registers contain physical addresses.

A process-space address translation that causes a translation-buffer miss will cause one memory reference for the process PTE. On an rtVAX, this reference is to physical memory, and cannot experience further memory-mapping problems.

If part or all of either process-space page table is mapped into I/O space, into nonexistent memory, or beyond the end of the physical address space while memory mapping is enabled, the operation of the processor is UNDEFINED.

The P0 region of the address space is mapped by the P0 page table (P0PT) which is defined by the P0 base register (P0BR) and the P0 length register (P0LR). The P0BR contains the base address of the P0PT, which in a (non-rtVAX) VAX is a virtual address in the system region, and in an rtVAX is a physical address. The page table entry addressed by the P0BR maps the first page of the P0

region of the virtual address space, that is, virtual byte address 0. The P0LR contains the size of the P0PT in longwords, that is, the number of page table entries.

The PTEs in the P0PT contain the mapping information, or point to the mapping information in the global page table if the PTE is in GPTX format. (See section 7.5.1.)

Writing P0LR bits ⟨26:24⟩ has no effect. P0LR bits ⟨26:24⟩ read as zero. At processor initialization time, the contents of both registers are UNPREDICTABLE.

The virtual page number is contained in bits ⟨29:9⟩ of the virtual address. A 22-bit length field is required to express the values 0 through 2^{21} inclusive. There could be as many as 2^{21} pages in the P0 region.

The algorithm an rtVAX uses to generate a physical address from a P0-region virtual address is as follows:

```
PROC_PA = (P0BR+4*P0VA<29:9>)<PFN>'P0VA<8:0>     ! P0 Region
```

The P1 region of the address space is mapped by the P1 page table (P1PT). P1PT is defined by the P1 base register (P1BR) and the P1 length register (P1LR). Because P1 space grows toward smaller addresses, and because a consistent hardware interpretation of the base and length registers is desirable, P1BR and P1LR describe the portion of P1 space that is not accessible. The P1BR contains the base address of the P1PT, which in a (non-rtVAX) VAX is a virtual address in the system region, and in an rtVAX is a physical address. Note that P1LR contains the number of nonexistent PTEs. P1BR contains the address of what would be the PTE for the first page of P1, that is, virtual byte address 40000000 (hex).

The PTEs in the P1PT contain the mapping information, or point to the mapping information in the GPT if the PTE is in GPTX format. (See section 7.5.1 for a description of the GPTX format.)

At processor initialization time, the contents of both registers are UNPREDICTABLE. Writing P1LR⟨31⟩ has no effect. The bit always reads as 0.

The algorithm an rtVAX uses to generate a physical address from a P1-region virtual address is as follows:

```
PROC_PA = (P1BR+4*P1VA<29:9>)<PFN>'P1VA<8:0>     ! P1 Region
```

11.2.4 *UNIBUS Interconnect*

If a UNIBUS interconnect is supported on a processor implementation, then all the following requirements for its support must be included.

- Interrupt levels 14 through 17 (hex) corresponding to the UNIBUS levels BR4 through BR7. (See sections 5.1, 5.6.2, and 5.9.2.)
- UNIBUS interrupt vectors in SCB pages. (See section 5.9.2.)
- One or more areas of UNIBUS I/O space, each 2^{18} bytes in length, that map to UNIBUS addresses. (See sections 7.5.2 and B.1.1.)
- Certain I/O-bus adapters need separate TB entries for each UNIBUS memory page they can map, and they pre-load all needed TB entries.
- Some byte and word references of read-modify-write type in UNIBUS I/O spaces (including ADAWI) interlock correctly using the DATIP and DATO(B) functions. (See sections 7.5.2, 7.1.3, and 9.13.)

11.3 *Instruction Emulation*

VAX processors and their operating systems cooperate to support emulation of those instructions that are omitted from the processor's instruction set. Programs running under the operating system can make use of these instructions as though they were supported directly by the processor.

The mechanism for emulating an omitted instruction depends on the instruction type. Emulation of instructions in the following groups is done entirely by software. Upon execution, a reserved-instruction fault is taken.

- extended-accuracy group
- compatibility mode group
- these floating-point instructions in the emulated-only group: ACB{F,D,G,H}, EMOD{F,D,G,H}, POLY{F,D,G,H}

- vector instructions (optional)

Emulation of string instructions in the following groups is assisted by the processor through the instruction-emulation exception

- packed-decimal-string group
- string instructions in the emulated-only group: MATCHC, MOVTC, MOVTUC, CRC, EDITPC

The process of emulating an omitted string instruction consists of the following steps:

1. The processor reads the instruction opcode and finds that this is an omitted instruction. The processor saves the opcode.
2. The processor evaluates the operand specifiers in order of instruction stream occurrence. The processor saves the operand address for each operand of write-access type or address type, and it reads and saves the operand itself for operands of read-access type.
3. The processor initiates an instruction-emulation trap. The current stack is probed. If the probe faults, the instruction faults, else, an emulation trap frame is pushed on the stack. The opcode and operands (or their addresses) are part of the trap frame. Unlike many exceptions, instruction-emulation trap does not cause the processor to enter kernel mode. The exception handler runs in the same mode as the trapped instruction, and the trap frame is pushed on the current stack.
4. The emulation-exception handler in the operating system examines the opcode of the trapped instruction and dispatches to the appropriate emulation routine.
5. The instruction-emulation routine reads and writes the instruction operands, as appropriate to the instruction being emulated. The operands need not be probed, since the emulation exception handler is running in the same mode as the emulated instruction.
6. The instruction-emulation routine sets the condition codes in the PSL on the stack, pops the emulation trap frame (except for the new PC and PSL) from the stack, and returns with REI.
7. Emulation is now complete, and the instruction following the emulated instruction begins execution.

If, during the emulation of an instruction, an exception such as access violation occurs, the emulation code must gain control,

save state in the registers just as the emulated instruction would, set FPD in the saved PSL, and reflect the exception to the user's current exception handler. If the conditions causing the exception are corrected and the exception was a fault, the instruction can be restarted. In this case, PSL⟨FPD⟩ will be set when instruction execution begins. Emulation consists of the following steps:

1. The processor reads the opcode and finds that this is an omitted instruction and that PSL⟨FPD⟩ is set.
2. The processor initiates a suspended-emulation fault. The current stack is probed. If the probe faults, then the instruction faults, else the processor pushes PC and PSL onto the stack.
3. The emulation-exception handler rebuilds the intermediate state of the instruction, using the information saved in the general registers at the time the emulated instruction was faulted.
4. The emulation handler resumes emulation of the instruction, as in steps 5 through 7 in the preceding list.

Emulation software runs in the mode of the emulated instruction and uses the same stack. Emulation software may allocate and use up to five pages of stack space for temporary storage. The contents of this area are UNPREDICTABLE after execution of an emulated instruction. If an emulated instruction addresses part of this area as an operand without first allocating it, or if an emulated instruction uses SP as an operand, the results of the instruction are UNPREDICTABLE. That is, the instructions *DIVF3 R1, R2, –50(SP)* and *DIVF3 R1, R2, SP* produce UNPREDICTABLE results. The instruction *DIVF3 R1, R2, –(SP)* allocates the area on top of the stack before using it and is legal.

11.3.1 Instruction-Emulation Exceptions

When a processor executes a string instruction that is omitted from its instruction set, an emulation exception results. An emulation exception occurs through one of two SCB vectors, depending on whether PSL⟨FPD⟩ is set at the beginning of the instruction. If PSL⟨FPD⟩ is clear, an instruction-emulation trap occurs through the SCB vector at offset C8 (hex), and an instruction-emulation trap frame is pushed on the stack. The PC pushed points to the instruction following the omitted instruction. If PSL⟨FPD⟩ is set, a suspended-emulation fault occurs through the SCB vector at offset CC (hex), and PC and PSL are pushed on the stack. The PC pushed points to the faulted instruction.

opcode	:(SP)
old PC	
specifier #1	
specifier #2	
specifier #3	
specifier #4	
specifier #5	
specifier #6	
specifier #7	
specifier #8	
new PC	
saved PSL	

Figure 11.1
Instruction-Emulation Trap Frame

In either case, if PSL⟨T⟩ is set at the time of the trap, PSL⟨TP⟩ is set in the PSL pushed on the stack. All other bits in the pushed PSL are unchanged. If PSL⟨FPD⟩ was set, it is set in the saved PSL.

The new PSL has ⟨TP,FPD,IV,DV,FU,T⟩ clear. All other fields are unchanged, including PSL⟨CUR_MOD,PRV_MOD,IS,IPL⟩. That is, the emulation-exception handler runs in the mode of the emulated instruction, on the same stack, and at the same IPL. The exception parameters are pushed on the current stack. (If the current stack cannot be written, the processor takes a memory management fault rather than an emulation exception.)

If either emulation-exception vector has bits ⟨1:0⟩ set to 1 (indicating that the exception is to be taken on the interrupt stack), the operation of the processor is UNDEFINED.

The emulation-exception stack frame is shown in Figure 11.1 and includes the following:

- Opcode — contains the opcode of the trapped instruction.
- Old PC — contains the address of the trapped instruction.
- Specifiers 1 through 8 — contain the addresses of corresponding instruction operands or the operands themselves. For each

operand of the trapped instruction, if the operand is of read access type (.rx), the parameter contains the operand value; if the operand is write access type (.wx) or address type (.ax), the parameter contains the operand address. For read-type operands of byte size, bits ⟨31:8⟩ of the longword are UNPREDICTABLE. For read-type operands of word size, bits ⟨31:16⟩ are UNPREDICTABLE. When an operand is in a register, the register is denoted by a reserved system space address corresponding to the one's complement of the register number. The parameter corresponding to an instruction operand that does not exist is UNPREDICTABLE. For example, if the trapped instruction has four operands, the parameters for specifiers 5 through 8 are UNPREDICTABLE.

- New PC — contains the address of the instruction following the trapped instruction.
- Saved PSL — contains the PSL at the time of the trap. If PSL⟨T⟩ was set at the beginning of the instruction, saved PSL⟨TP⟩ is set.

12

VAX Vector Registers and Instructions

12.1 Introduction

This chapter and Chapter 13 describe an extension to the VAX architecture for integrated vector processing. This chapter describes the vector registers and vector instructions of the VAX Vector Architecture. Chapter 13 describes the execution model and exception reporting for the vector architecture.

Some of the VAX vector architecture departs from the traditional VAX scalar architecture, especially in the areas of UNPREDICTABLE results, vector processor exceptions, and the synchronization of instructions and memory.

Implementation of the VAX vector architecture is optional. VAX processors that do implement the vector architecture do so as specified in these two chapters. Operating system software may emulate the vector architecture on processors that omit it.

On VAX processors that omit the vector architecture, vector instructions result in a reserved-instruction fault.

The vector architecture features include additional instructions, vector registers, and vector control registers.

All examples of vector instructions follow the assembler notation specified at the end of this chapter. The number and order of oper-

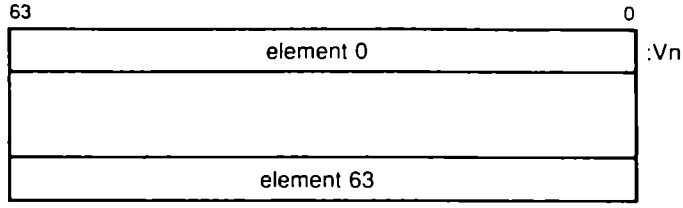


Figure 12.1
Vector Register

ands for the assembler notation is different from the instruction-stream format specified in the vector instruction descriptions.

12.2 Registers

A VAX vector processor comprises a number of different kinds of registers. Sixteen vector registers are used to contain as many as 1024 elements of vector data. The operation of vector instructions on these 16 vector registers is controlled by a series of vector control registers. Finally, the operation and status of the vector processor as a whole are handled by vector internal processor registers (IPRs). All of these are described in the following sections.

12.2.1 Vector Registers

There are 16 vector registers, V0 through V15. Each vector register contains 64 elements, numbered 0 through 63. Each element is 64 bits wide. Figure 12.1 depicts a vector register.

A vector instruction that performs a register-to-register operation is defined as a "vector operate instruction." A vector operate instruction that reads or writes F_floating data, integer data for shifts, or integer arithmetic operations reads bits ⟨31:0⟩ of each source element and writes bits ⟨31:0⟩ of each destination element. Bits ⟨63:32⟩ of the destination are UNPREDICTABLE for F_floating, integer arithmetic, and shift instructions.

Vector logical instructions read bits ⟨31:0⟩ of each source element and write the result into bits ⟨31:0⟩ of each destination element; bits ⟨63:32⟩ of the destination element receive bits ⟨63:32⟩ of the corresponding element of the Vb source operand.

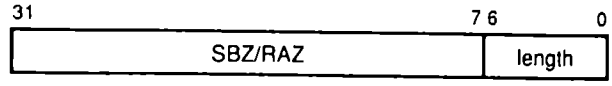


Figure 12.2
Vector Length Register (VLR)

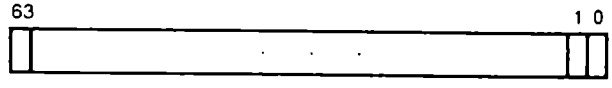


Figure 12.3
Vector Mask Register (VMR)

For vector instructions that read longword data from memory into a vector register (VLDL and VGATHL), bits ⟨63:32⟩ of the destination elements are UNPREDICTABLE.

If the same vector register is used as both source and destination in a VGATH instruction, the result of the VGATH is UNPREDICTABLE.

For the IOTA vector instruction, bits ⟨63:32⟩ of the destination elements are UNPREDICTABLE.

12.2.2 *Vector Control Registers*

The 7-bit vector length register (VLR) limits the highest vector element to be processed by a vector instruction. The VLR is loaded prior to executing the vector instruction using a MTVP instruction. The value in the VLR may range from 0 to 64. If the vector length is zero, no vector elements are processed. If a vector instruction is executed with vector length greater than 64, its results are UNPREDICTABLE. Elements beyond the vector length in the destination vector register are not modified. See Figure 12.2.

The vector mask register (VMR) has 64 bits, each corresponding to an element of a vector register. Bit ⟨0⟩ corresponds to vector element 0. See Figure 12.3.

The 7-bit vector count register (VCR) receives the length of the offset vector generated by the IOTA instruction. See Figure 12.4.

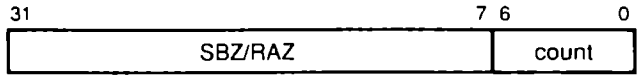


Figure 12.4
Vector Count Register (VCR)

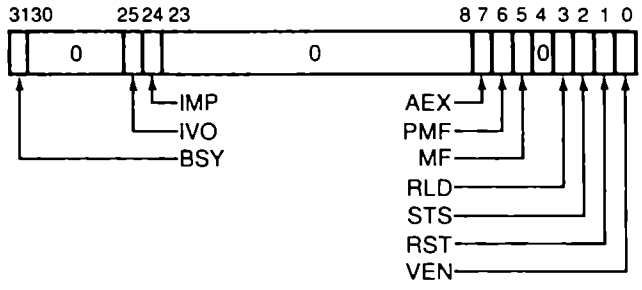


Figure 12.5
Vector Processor Status Register (VPSR)

These registers are read and written by move from/to vector processor (MFVP/MTVP) instructions.

12.2.3 *Internal Processor Registers*

The vector processor contains the following IPRs that can be accessed by the scalar processor using MTPR/MFPR instructions:

- Vector processor status register (VPSR)
- Vector arithmetic exception register (VAER)
- Vector memory activity check (VMAC)
- Vector translation buffer invalidate all (VTBIA)
- Vector state address register (VSAR)

The VPSR is shown in Figure 12.5, and described in Table 12.1. Table 12.2 shows the possible settings of VPSR⟨3:0⟩ in the same MTPR instruction and the resulting action for the vector processor. The state of the vector processor is determined by the encoding of the vector processor enabled (VPSR⟨VEN⟩) and vector processor busy (VPSR⟨BSY⟩) bits. The vector processor state for possible encodings is shown in Table 12.3. Note that because the vector and scalar processors can execute asynchronously, a VPSR state transition may not be seen immediately by the scalar processor. After performing an MTPR to the VPSR, software must then issue an MFPR from the VPSR to ensure that the new state of

Table 12.1
Description of the Vector Processor Status Register (Part 1)

Extent	Type	Description
⟨0⟩	R/W	Vector processor enabled (VEN). The vector processor is enabled by writing a 1 to this bit. Writing a 0 disables the vector processor. If VPSR⟨VEN⟩ is cleared by software while VPSR⟨BSY⟩ is set, then once the new state of the VPSR becomes synchronized with subsequent vector instructions, no more are sent and the vector processor completes execution of all pending instructions in its instruction queue. See section 13.2.3 for more details.
⟨1⟩	W	Vector processor state reset (RST). Writing a 1 to this bit clears VPSR and VAER. If VPSR⟨RST⟩ is set by software while VPSR⟨BSY⟩ is set, the operation of the vector processor is UNDEFINED. This bit is RAZ.
⟨2⟩	W	Vector state store (STS). Writing a 1 to this bit initiates storing of implementation-specific vector state information to memory using the address in the VSAR for the asynchronous method of handling memory management exceptions. If the synchronous method is implemented, writes to VPSR⟨STS⟩ are ignored. This bit is RAZ.
⟨3⟩	W	Vector state reload (RLD). Writing a 1 to this bit initiates reloading of implementation-specific vector state information from memory using the address in VSAR for the asynchronous method of handling memory management exceptions. If the synchronous method is implemented, writes to VPSR⟨RLD⟩ are ignored. This bit is RAZ.
⟨4⟩	R	0
⟨5⟩	R/W1C	Memory fault (MF). This bit is set by the vector processor when there is a memory reference to be retried due to an asynchronous memory management exception. Writing a 1 to VPSR⟨MF⟩ clears it. Writing a 0 to VPSR⟨MF⟩ has no effect. If the synchronous method of handling memory management exceptions is implemented, this bit is always 0.

the VPSR (and the VAER if cleared by VPSR⟨RST⟩) will affect the execution of subsequently issued vector instructions. The MFPR in this case will not complete until the new state of the vector processor becomes visible to the scalar processor. If software does not issue the MFPR, then it is UNPREDICTABLE whether this synchronization between the new state of the VPSR (and the VAER) and subsequently issued vector instructions occurs.

The VAER is shown in Figure 12.6. This read-only register is used to record information regarding vector arithmetic exceptions. Table 12.4 shows the encoding for the exception condition types. The destination register mask field of the VAER records which vector registers have received default results due to arithmetic exceptions. VAER⟨16+n⟩ corresponds to vector register Vn, where

Table 12.1
Description of the Vector Processor Status Register (Part 2)

Extent	Type	Description
⟨6⟩	R/W1C	Pending memory fault (PMF). This bit is set by the vector processor when an asynchronous memory management exception is pending. Writing a 1 to VPSR⟨PMF⟩ clears it. Writing a 0 to VPSR⟨PMF⟩ has no effect. If the synchronous method of handling memory management exceptions is implemented, this bit is always 0.
⟨7⟩	R/W1C	Vector arithmetic exception (AEX). This bit is set by the vector processor when disabling itself due to an arithmetic exception. Information regarding the nature of the exception can be found in the VAER. Writing a 1 to VPSR⟨AEX⟩ clears VPSR⟨AEX⟩ and the VAER. Writing a 0 to VPSR⟨AEX⟩ has no effect.
⟨23:8⟩	R	0
⟨24⟩	R/W1C	Implementation-specific hardware error (IMP). This bit is set by the vector processor when disabling itself due to an implementation-specific hardware error. Writing a 1 to VPSR⟨IMP⟩ clears it. Writing a 0 to VPSR⟨IMP⟩ has no effect.
⟨25⟩	R/W1C	Illegal vector opcode (IVO). This bit is set by the vector processor when disabling itself due to receiving an illegal vector opcode. Writing a 1 to VPSR⟨IVO⟩ clears it. Writing a 0 to VPSR⟨IVO⟩ has no effect.
⟨30:26⟩	R	0
⟨31⟩	R	Vector processor busy (BSY). When this bit is set, the vector processor is executing vector instructions. When it is clear, the vector processor is idle, or the vector processor has suspended instruction execution due to an asynchronous memory management exception or hardware error. Writing to VPSR⟨BSY⟩ has no effect.

n is between 0 and 15. For more information, refer to section 13.2.2.

The VMAC register is a read-only register and is used to guarantee the completion of all prior vector memory accesses. For more information on this function of the VMAC, refer to section 13.3.2. An MFPR from the VMAC also ensures that all hardware errors encountered by previous vector memory instructions are reported before the MFPR completes. For more information on this function of the VMAC, refer to section 13.5. The value returned by MFPR from the VMAC is UNPREDICTABLE.

The VTBIA register is a write-only register and may be omitted in some implementations. If the vector processor contains its own translation buffer, moving 0 into the VTBIA using the MTPR

Table 12.2
Possible VPSR Settings for MTPR

RLD	STS	RST	VEN	Meaning
0	0	0	0	Disable vector processor
0	0	0	1	Enable vector processor
0	0	1	0	Reset state and disable vector processor
0	0	1	1	Reset state and enable vector processor
0	1	0	0	Store state (must disable vector processor)
1	0	0	0	Reload state and disable vector processor
1	0	0	1	Reload state and then enable vector processor

Table 12.3
State of the Vector Processor

VEN	BSY	Meaning
0	0	The vector processor is not executing any instructions now, and either has no pending instructions or will not execute pending instructions. No more instructions should be sent.
0	1	The vector processor is executing at least one pending instruction. No more instructions should be sent.
1	0	The vector processor is not executing any instructions now, and either has no pending instructions or will not execute pending instructions. New instructions can be sent to the vector processor.
1	1	The vector processor is executing at least one instruction now. New instructions can be sent.

instruction invalidates the entire vector translation buffer. For more information, refer to section 13.4.

The VSAR is shown in Figure 12.7. This read/write register contains a quadword-aligned virtual address of memory assigned by software for storing implementation specific vector hardware state when the asynchronous method of handling memory man-

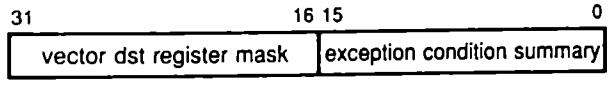


Figure 12.6
Vector Arithmetic Exception Register (VAER)

Table 12.4
VAER Exception Condition Summary Word Encoding

Bit	Exception Condition
⟨0⟩	Floating Underflow
⟨1⟩	Floating Divide by Zero
⟨2⟩	Floating Reserved Operand
⟨3⟩	Floating Overflow
⟨4⟩	0
⟨5⟩	Integer Overflow
⟨15:6⟩	0

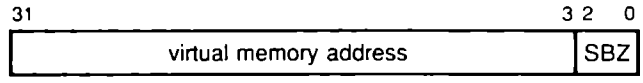


Figure 12.7
Vector State Address Register (VSAR)

agement exceptions is implemented. The length of this memory area is implementation-specific. Software must guarantee that accessing the memory pointed to by the address does not result in a memory management exception. If the synchronous method of handling memory management exceptions is implemented, this register is omitted. For more information, refer to section 13.2.1.

With the exception of the VPSR (and the VAER), an MTPR to any other writable vector IPR ensures that the new state of the IPR affects the execution of all subsequently issued vector instructions. Vector instructions issued before an MTPR to any writable vector IPR are unaffected by the new state of the IPR (and any implicitly changed vector IPR) except in one case: when the MTPR sets VPSR⟨RST⟩ while VPSR⟨BSY⟩ is set. (See Table 12.1 for more details.)

Except for the cases listed below, the operations of the scalar and vector processors are UNDEFINED after execution of an MTPR to a read-only vector IPR, an MTPR to a non-existent vector IPR, an MTPR of a non-zero value to a MBZ field, or an MTPR of a

Table 12.5
IPR Assignments

Offset (Hex)	IPR
90	VPSR
91	VAER
92	VMAC
93	VTBIA
94	VSAR
95–9B	Reserved for vector architecture use
9C–9F	Reserved for vector implementation use

reserved value to a vector IPR. The preferred implementation is to cause reserved-operand fault.

- If an implementation supports an optional vector processor, but the vector processor is not installed, an MTPR to the VPSR has no effect.
- If an implementation supports an optional vector processor, but the vector processor is not installed or the scalar/vector processor pair uses a common translation buffer, an MTPR to the VTBIA has no effect.

In each of these cases, MTPR is implemented as a no-op.

Except for the cases listed below, the operations of the scalar and vector processors are UNDEFINED after execution of an MFPR from a non-existent vector IPR or of an MFPR from a write-only vector IPR. The preferred implementation is to cause reserved-operand fault.

- If an implementation supports an optional vector processor, but the vector processor is not installed, an MFPR from the VPSR returns 0.
- If an implementation supports an optional vector processor, but the vector processor is not installed, an MFPR from the VMAC has no effect.

The IPR assignments for these registers are found in Table 12.5.

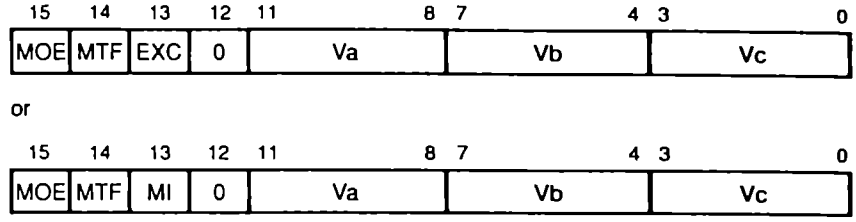


Figure 12.8
Vector Control Word Operand Format (cntrl)

12.3 *Vector Instruction Formats*

Vector instructions use two-byte opcodes and normal VAX operand specifiers. (For more information on VAX operand specifiers, refer to Chapter 2.) The vector registers to be used by a vector instruction are specified by the vector control word operand. The MFVP, MTVP, and VSYNC instructions do not use a vector control word operand. The general format of the vector control word operand is shown in Figure 12.8. Table 12.6 describes the fields of the vector control word operand (cntrl). The actual format of the vector control word operand is instruction dependent. (Refer to the instruction descriptions later in this chapter for more detail.) The vector control word operand is passed by the VAX scalar processor to the vector processor.

The vector control word operand may determine some or all of the following:

- Enabling of masked operations
- Enabling of floating underflow for floating-point instructions and integer overflow for integer operations
- Which vector registers to use as sources/destinations
- Which type of operation to perform (for the convert and compare instructions)

12.3.1 *Masked Operations*

Masked operations are enabled by the use of cntrl⟨15:14⟩ of the vector control word operand. Cntrl⟨15⟩ is the masked operations enable bit (MOE), and cntrl⟨14⟩ is the match true/false bit (MTF). When cntrl⟨MOE⟩ is set, masked operations are enabled. Only elements for which the corresponding vector mask register (VMR) bit matches cntrl⟨MTF⟩ are operated on. If cntrl⟨MOE⟩ is clear, all

Table 12.6
Description of the Vector Control Word Operand

Extent	Name	Description
⟨15⟩	MOE	Masked operation enable. This bit enables operations under the control of the vector mask register (VMR) for vector instructions. When set, masked operations are enabled, and only elements for which the corresponding VMR bit matches cntrl⟨MTF⟩ are operated on. If cntrl⟨MOE⟩ is clear, all elements are operated on. In either case, the Vector Length Register (VLR) limits the highest element operated on.
⟨14⟩	MTF	Match true/false. When masked operations have been enabled (cntrl⟨MOE⟩ EQL 1), only elements for which the corresponding VMR bit matches cntrl⟨MTF⟩ are operated on. See above description. Cntrl⟨MTF⟩ is also used by the VMERGE and IOTA instructions.
⟨13⟩	EXC	Exception enable. Used only in vector integer and floating-point instructions to enable integer overflow and floating underflow, respectively.
⟨13⟩	MI	Modify intent. Used only in VLD/VGATH instructions to indicate that a majority of the memory locations being loaded by the VLD or VGATH later will be stored into by VST/VSCAT instructions. This bit is optional to implement. See section 12.3.3.
⟨12⟩		SBZ (should be zero).
⟨11:8⟩	Va	This field selects the vector register to be used as the Va operand. For VVCVT, it specifies the convert function.
⟨7:4⟩	Vb	This field selects the vector register to be used as the Vb operand.
⟨3:0⟩	Vc	This field selects the vector register to be used as the Vc operand. For VCMP, it specifies the compare function.

elements are operated on. In either case, the vector length register (VLR) limits the highest element operated on.

Cntrl⟨MOE⟩ should be 0 for VMERGE and IOTA instructions; otherwise the results are UNPREDICTABLE. Both the VMR and cntrl⟨MTF⟩ are always used by these instructions. VMERGE and IOTA operate on vector register elements up to the value specified in VLR.

12.3.2 *Exception Enable Bit*

The vector processor does not use the IV and FU bits in the PSL to enable integer overflow and floating underflow exception con-

ditions. These exception conditions are enabled or disabled on a per-instruction basis for vector integer and floating-point instructions by bit ⟨13⟩ in the vector control word operand (cntrl⟨EXC⟩). When cntrl⟨EXC⟩ is set, floating underflow is enabled for vector floating point instructions, and integer overflow is enabled for vector integer instructions. When cntrl⟨EXC⟩ is clear, floating underflow and integer overflow are disabled.

Note that for VLD/VGATH instructions, bit⟨13⟩ is used and labeled differently.

12.3.3 *Modify Intent Bit*

By using the modify intent bit, software can indicate to the vector processor that a majority of the memory locations being loaded by a VLD/VGATH instruction later will be stored into by VST/VSCAT instructions and so become modified. When informed of software's intent to modify, some vector processor implementations can optimize the vector loads and stores performed on these locations.

The modify intent bit resides in bit⟨13⟩ of the vector control word operand (cntrl⟨MI⟩) and is used only in VLD and VGATH instructions. A vector processor implementation is not required to implement cntrl⟨MI⟩.

For vector processors that implement cntrl⟨MI⟩, software uses the bit in a VLD or VGATH instruction in the following way:

- By setting cntrl⟨MI⟩ to 0, software indicates that less than a majority of the locations loaded by the VLD/VGATH later will be stored into by VST/VSCAT instructions.
- By setting cntrl⟨MI⟩ to 1, software indicates that a majority of the locations loaded by the VLD/VGATH will later be stored into by VST/VSCAT instructions.

Vector processors that do not implement cntrl⟨MI⟩ ignore the setting of this bit in the control word for VLD and VGATH.

The results of VLD/VGATH and VST/VSCAT are unaffected by the setting of cntrl⟨MI⟩. This includes memory management, where access checking is done with read intent for VLD/VGATH even if cntrl⟨MI⟩ is set. However, incorrectly setting cntrl⟨MI⟩ can prevent the optimization of these instructions.

15	14	13	12	11 … 8	7 … 4	3 … 0
MOE	MTF	MI	0	0	0	Dst

VLDL and VLDQ

15	14	13	12	11 … 8	7 … 4	3 … 0
MOE	MTF	0	0	0	0	Src

VSTL and VSTQ

15	14	13	12	11 … 8	7 … 4	3 … 0
MOE	MTF	MI	0	0	Src	Dst

VGATHL and VGATHQ

15	14	13	12	11 … 8	7 … 4	3 … 0
MOE	MTF	0	0	0	Src2	Src1

VSCATL and VSCATQ

15	14	13	12	11 … 8	7 … 4	3 … 0
MOE	MTF	EXC	0	Src1	Src2	Dst

VVADDL/F/D/G, VVSUBL/F/D/G, VVMULL/F/D/G, and VVDIVF/D/G

Figure 12.9
Vector Control Word Formats (Part 1)

12.3.4 *Register Specifier Fields*

The Va (cntrl⟨11:8⟩), Vb (cntrl⟨7:4⟩), and Vc (cntrl⟨3:0⟩) fields of the vector control word operand are generally used to select vector registers. Some vector instructions use these fields to encode other instruction-specific information, as shown later in this section.

12.3.5 *Vector Control Word Formats*

Depending on the instruction, the vector control word can specify up to two vector registers as sources and one vector register as a destination. Other information may be encoded in the vector control word, as shown in Figure 12.9. Bits that are shown as "0" should be zero (SBZ). Execution of vector instructions with illegal, inconsistent, or unspecified control word fields produces UNPREDICTABLE results.

15	14	13	12	11 – 8	7 – 4	3 – 0
MOE	MTF	0	0	Src1	Src2	Dst

VVSLLL, VVSRLL, VVBISL, VVXORL, and VVBICL

15	14	13	12	11 – 8	7 – 4	3 – 0
MOE	MTF	0	0	Scr1	Scr2	cmp func

VVCMPL/F/D/G

15	14	13	12	11 – 8	7 – 4	3 – 0
MOE	MTF	EXC	0	cvt func	Src	Dst

VVCVT

15	14	13	12	11 – 8	7 – 4	3 – 0
0	MTF	0	0	Scr1	Scr2	Dst

VVMERGE

15	14	13	12	11 – 8	7 – 4	3 – 0
MOE	MTF	EXC	0	0	Src	Dst

VSADDL/F/D/G, VSSUBL/F/D/G, VSMULL/F/D/G, and VSDIVF/D/G

Figure 12.9
Vector Control Word Formats (Part 2)

Figure 12.9 depicts the vector control word for VVSLLL, VVSRLL, VVBISL, VVXORL, and VVBICL. Cntrl⟨EXC⟩ should always be 0 for these instructions, otherwise the results are UNPREDICTABLE.

Figure 12.9 depicts the vector control word for VVCMPL/F/D/G. The Vc field (cntrl⟨3:0⟩) is used to specify the compare function.

Figure 12.9 depicts the vector control word for VVCVT. The Va field (cntrl ⟨11:8⟩) is used to specify the convert function.

Figure 12.9 depicts the vector control word for VSSLLL, VSSRLL, VSBISL, VSXORL, and VSBICL. Cntrl⟨EXC⟩ should be zero for these instructions; otherwise, the results are UNPREDICTABLE.

15	14	13	12	11 … 8	7 … 4	3 … 0
MOE	MTF	0	0	0	Src	Dst

VSSLLL, VSSRLL, VSBISL, VSXORL, and VSBICL

15	14	13	12	11 … 8	7 … 4	3 … 0
MOE	MTF	0	0	0	Src	cmp func

VSCMPL/F/D/G

15	14	13	12	11 … 8	7 … 4	3 … 0
0	MTF	0	0	0	Src	Dst

VSMERGE

15	14	13	12	11 … 8	7 … 4	3 … 0
0	MTF	0	0	0	0	Dst

IOTA

Figure 12.9
Vector Control Word Formats (Part 3)

Figure 12.9 depicts the vector control word for VSCMPL/F/D/G. The Vc field (cntrl⟨3:0⟩) is used to specify the compare function.

12.3.6 *Restrictions on Operand Specifier Usage*

Certain restrictions are placed on the addressing mode combinations used in a single vector instruction. These combinations involve the logically inconsistent simultaneous use of a value as both a source operand (i.e., an .rw, .rl, or .rq operand) and an address. Specifically, if in the same instruction the contents of register Rn are used as both a part of a source operand and as an address in an addressing mode that modifies Rn (i.e., autodecrement, autoincrement, or autoincrement deferred), the value of the scalar source operand is UNPREDICTABLE.

Use of short literal mode for the scalar source operand of a vector floating-point instruction causes UNPREDICTABLE results.

If a VST or VSCAT instruction overwrites anything needed for calculation of the memory addresses to be written, the result of the VST or VSCAT is UNPREDICTABLE.

If the same vector register is used as both source and destination in a VGATH instruction, the result of the VGATH is UNPREDICTABLE.

When the addressing mode of the BASE operand used in a VLD, VST, VGATH, or VSCAT instruction is immediate, the results of the instruction are UNPREDICTABLE.

12.3.7 *VAX Condition Codes*

The vector instructions do not affect the condition codes in the PSL of the associated scalar processor.

12.3.8 *Illegal Vector Opcodes*

An illegal vector opcode is defined as a vector opcode to which no vector processor function is currently assigned. Opcodes that are not listed in Table A.3 as vector opcodes are neither decoded nor executed by the vector processor.

An implementation is permitted to report an illegal vector opcode in one of two ways:

1. *Reserved-instruction fault.* This is the recommended implementation.
2. *Illegal vector opcode.* The vector processor disables itself and sets VPSR⟨IVO⟩. The remainder of the vector processor state is left unmodified.

How a particular illegal vector opcode is reported is implementation specific.

12.4 *Vector Instruction Classes*

The vector instruction set is divided into six major sections:

- Vector memory
- Vector integer
- Vector logical and shift
- Vector floating point
- Vector edit
- Miscellaneous operations

Within each section, closely related instructions are grouped and described together using the same conventions as established in Chapter 3.

12.5 Vector Memory Access Instructions

Considerations for the vector memory access instructions include alignment, stride, address specifier context, and access mode.

These instructions require their vector operands to be naturally aligned in memory. Longwords must be aligned on longword boundaries. Quadwords must be aligned on quadword boundaries. If any vector element is not naturally aligned in memory, an access control violation occurs. For further details, see section 13.2.1. The scalar operands need not be naturally aligned in memory.

A vector's stride is defined as the number of memory locations (bytes) between the starting address of consecutive vector elements. A contiguous vector that has longword elements has a stride of 4; a contiguous vector that has quadword elements has a stride of 8.

The base address specifier used by the vector memory access instructions is of byte context, regardless of the data type. Arrays are addressed as byte strings. Index values in array specifiers are multiplied by 1, and the amount of autoincrement or autodecrement, when either of these modes is used, is 1.

A vector memory access instruction is executed using the access mode in effect at the time the instruction is issued by the scalar processor.

VLD Load Memory Data into Vector Register

Format

```
opcode cntrl.rw, base.ab, stride.rl
```

Vector Control Word

15	14	13	12	11 — 8	7 — 4	3 — 0
MOE	MTF	MI	0	0	0	Vc

Operation

```
addr ← base

FOR i ← 0 TO VLR-1
   BEGIN
   IF {{MOE EQL 0} OR
      {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
      BEGIN
      IF {addr unaligned} THEN
         {Vector Alignment Exception}
      IF VLDL THEN
         Vc[i] ← (addr)<31:0>
      IF VLDQ THEN
         Vc[i] ← (addr)<63:0>
      END
   addr ← addr + stride                    !Increment by stride
   END
```

VP Exceptions

```
Access Control Violation
Translation Not Valid
Vector Alignment
```

Opcodes

34FD	VLDL	Load Longword Vector from Memory to Vector Register
36FD	VLDQ	Load Quadword Vector from Memory to Vector Register

Description

The source operand vector is fetched from memory and written to vector destination register Vc. The length of the vector is specified by the VLR. The virtual address of the source vector is computed using the base address and the stride. The address of element i (0 LEQU i LEQU (VLR–1)) is computed as base+(i*stride). The stride can be positive, negative, or zero.

In VLDL, bits ⟨31:0⟩ of each destination vector element receive the memory data and bits ⟨63:32⟩ are UNPREDICTABLE.

If any vector element operated on is not naturally aligned in memory, a vector alignment exception occurs.

The results of VLD are unaffected by the setting of cntrl⟨MI⟩. For more details about the use of cntrl⟨MI⟩, see section 12.3.3.

If the addressing mode of the BASE operand is immediate, the results of the instruction are UNPREDICTABLE.

An implementation may load the elements of the vector in any order and more than once. When a vector processor memory management exception occurs, the contents of the destination vector elements are UNPREDICTABLE.

VGATH Gather Memory Data into Vector Register

Format

```
opcode cntrl.rw, base.ab
```

Vector Control Word

15	14	13	12	11 – 8	7 – 4	3 – 0
MOE	MTF	MI	0	0	Vb	Vc

Operation

```
FOR i ← 0 TO VLR-1
   BEGIN
   addr ← base + Vb[i]<31:0>
   IF {{MOE EQL 0} OR
      {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
      BEGIN
      IF {addr unaligned} THEN
         {Vector Alignment Exception}
      IF VGATHL THEN
         Vc[i] ← (addr)<31:0>
      IF VGATHQ THEN
         Vc[i] ← (addr)<63:0>
      END
   END
```

VP Exceptions

```
Access Control Violation
Translation Not Valid
Vector Alignment
```

Opcodes

35FD	VGATHL	Gather Longword Vector from Memory to Vector Register
37FD	VGATHQ	Gather Quadword Vector from Memory to Vector Register

Description

The source operand vector is fetched from memory and written to vector destination register Vc. The length of the vector is specified by VLR. The virtual address of the vector is computed using the base address and the 32-bit offsets in vector register Vb. The address of element i (0 LEQU i LEQU (VLR–1)) is computed as base+Vb[i]. The 32-bit offset can be positive, negative, or zero.

In VGATHL, bits ⟨31:0⟩ of each destination vector element receive the memory data, and bits ⟨63:32⟩ are UNPREDICTABLE.

If any vector element operated on is not naturally aligned in memory, a vector alignment exception occurs.

The results of VGATH are unaffected by the setting of cntrl⟨MI⟩. For more details about the use of cntrl⟨MI⟩, see section 12.3.3.

If the addressing mode of the BASE operand is immediate, the results of the instruction are UNPREDICTABLE.

An implementation may load the elements of the vector in any order and more than once. When a vector processor memory management exception occurs, the contents of the destination vector elements are UNPREDICTABLE.

If the same vector register is used as both source and destination, the result of the VGATH is UNPREDICTABLE.

VST Store Vector Register Data into Memory

Format

```
opcode cntrl.rw, base.ab, stride.rl
```

Vector Control Word

15	14	13	12	11 – 8	7 – 4	3 – 0
MOE	MTF	0	0	0	0	Vc

Operation

```
addr ← base

FOR i ← 0 TO VLR-1
   BEGIN
   IF {{MOE EQL 0} OR
      {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
      BEGIN
      IF {addr unaligned} THEN
         {Vector Alignment Exception}
      IF VSTL THEN
         (addr)<31:0> ← Vc[i]<31:0>
      IF VSTQ THEN
         (addr)<63:0> ← Vc[i]
      END
   addr ← addr + stride                    !Increment by stride
   END
```

VP Exceptions

```
Access Control Violation
Translation Not Valid
Vector Alignment
Modify
```

Opcodes

9CFD	VSTL	Store Longword Vector from Vector Register to Memory
9EFD	VSTQ	Store Quadword Vector from Vector Register to Memory

Description

The source operand in vector register Vc is written to memory. The length of the vector is specified by the VLR. The virtual address of the destination vector is computed using the base address and the stride. The address of element i (0 LEQU i LEQU (VLR–1)) is computed as base+(i*stride). The stride can be positive, negative, or zero.

If any vector element operated on is not naturally aligned in memory, a vector alignment exception occurs.

For a non-zero stride value, an implementation may store the vector elements in parallel; therefore, the order in which these elements are stored is UNPREDICTABLE. Furthermore, if the

non-zero stride causes result locations in memory to overlap, then the values stored in the overlapping result locations are also UNPREDICTABLE.

For a stride value of zero, the highest-numbered register element destined for the single memory location becomes the final value of that location.

When a vector processor memory management exception occurs, it is UNPREDICTABLE whether the vector processor writes any result location for which an exception did not occur. If the fault condition can be eliminated by software and the instruction restarted, then the vector processor will ensure that all destination locations are written.

If the destination vector overlaps the vector instruction control word, base, or stride operand, the result of the instruction is UNPREDICTABLE.

If the addressing mode of the BASE operand is immediate, the results of the instruction are UNPREDICTABLE.

VSCAT Scatter Vector Register Data into Memory

Format

```
opcode cntrl.rw, base.ab
```

Vector Control Word

15	14	13	12	11 – 8	7 – 4	3 – 0
MOE	MTF	0	0	0	Vb	Vc

Operation

```
FOR i ← 0 TO VLR-1
   BEGIN
   addr ← base + Vb[i]<31:0>
   IF {{MOE EQL 0} OR
      {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
      BEGIN
      IF {addr unaligned} THEN
         {Vector Alignment Exception}
      IF VSCATL THEN
         (addr)<31:0> ← Vc[i]<31:0>
      IF VSCATQ THEN
         (addr)<63:0> ← Vc[i]
      END
   END
```

VP Exceptions

```
Access Control Violation
Translation Not Valid
Vector Alignment
Modify
```

Opcodes

9DFD	VSCATL	Scatter Longword Vector from Vector Register to Memory
9FFD	VSCATQ	Scatter Quadword Vector from Vector Register to Memory

Description

The source vector operand Vc is written to memory. The length of the vector is specified by the VLR. The virtual address of the destination vector is computed using the base address operand and the 32-bit offsets in vector register Vb. The address of element i (0 LEQU i LEQU (VLR–1)) is computed as base+Vb[i]. The 32-bit offset can be positive, negative, or zero.

If any vector element operated on is not naturally aligned in memory, a vector alignment exception occurs.

An implementation may store the vector elements in parallel; therefore, the order in which elements are stored to different memory locations is UNPREDICTABLE. In the case where multiple elements are destined for the same memory location, the

highest-numbered element among them becomes the final value of that location.

When a vector processor memory management exception occurs, it is UNPREDICTABLE whether the vector processor writes any result location for which an exception did not occur. If the fault condition can be eliminated by software and the instruction restarted, then the vector processor will ensure that all destination locations are written.

If the destination vector overlaps the vector instruction control word or base operand, the result of the instruction is UNPREDICTABLE.

If the addressing mode of the BASE operand is immediate, the results of the instruction are UNPREDICTABLE.

12.6 *Vector Integer Instructions*

This class of instructions performs integer operations on vectors of longwords. The first source operand can be either an integer scalar value or a vector register containing longword data; the second source operand is always a vector register containing longword data.

VADDL Vector Integer Add

Format

```
opcode cntrl.rw                  ! Vector + Vector
opcode cntrl.rw, addend.rl       ! Scalar + Vector
```

Vector Control Word

15	14	13	12	11 – 8	7 – 4	3 – 0
MOE	MTF	EXC	0	Va or 0	Vb	Vc

Operation

```
FOR i ← 0 TO VLR-1
   IF {{MOE EQL 0} OR
      {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
      BEGIN
      IF VVADDL THEN
         Vc[i]<31:0> ← Va[i]<31:0> + Vb[i]<31:0>
      IF VSADDL THEN
         Vc[i]<31:0> ← addend + Vb[i]<31:0>
      END
```

VP Exceptions

Integer Overflow

Opcodes

80FD	VVADDL	Vector Vector Add Longword
81FD	VSADDL	Vector Scalar Add Longword

Description

The scalar addend or Va operand is added, element-wise, to vector register Vb, and the 32-bit sum is written to vector register Vc. Only bits ⟨31:0⟩ of each vector element participate in the operation. Bits ⟨63:32⟩ of the elements of vector register Vc are UNPREDICTABLE. The length of the vector is specified by the VLR.

If integer overflow is detected and cntrl⟨EXC⟩ is set, the exception type and destination register number are recorded in the VAER, and the vector operation is allowed to complete. On integer overflow, the low-order 32 bits of the true result are stored in the destination element.

VCMPL Vector Integer Compare

Format

```
opcode cntrl.rw                  ! Vector - Vector
opcode cntrl.rw, src.rl          ! Scalar - Vector
```

Vector Control Word

15	14	13	12	11 – 8	7 – 4	3 – 0
MOE	MTF	0	0	Va or 0	Vb	cmp func

Operation

```
CASE cntrl<3:0> OF
   0:          SIGNED_RELATION is Greater Than;
   1:          SIGNED_RELATION is Equal;
   2:          SIGNED_RELATION is Less Than;
   4:          SIGNED_RELATION is Less Than or Equal;
   5:          SIGNED_RELATION is Not Equal;
   6:          SIGNED_RELATION is Greater Than or Equal;
   otherwise: SIGNED_RELATION is Reserved;
   END

FOR i ← 0 TO VLR-1
   IF {{MOE EQL 0} OR
      {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
      BEGIN
      IF VVCMPL THEN
         IF Va[i]<31:0> SIGNED_RELATION Vb[i]<31:0> THEN
            VMR<i> ← 1
         ELSE
            VMR<i> ← 0
      IF VSCMPL THEN
         IF src SIGNED_RELATION Vb[i]<31:0> THEN
            VMR<i> ← 1
         ELSE
            VMR<i> ← 0
      END
```

VP Exceptions none

Opcodes

C0FD	VVCMPL	Vector Vector Compare Longword
C1FD	VSCMPL	Vector Scalar Compare Longword

Description

The scalar or Va operand is compared, element-wise, with vector register Vb. The length of the vector is specified by the VLR. For each element comparison, if the specified relationship is true, the vector mask register bit (VMR⟨i⟩) corresponding to the vector element is set to 1; otherwise, it is cleared. If cntrl⟨MOE⟩ is set, the VMR bits corresponding to elements that do not match cntrl⟨MTF⟩ are left unchanged. VMR bits beyond the vector length

are left unchanged. Only bits ⟨31:0⟩ of each vector element participate in the operation.

Vector integer compare instructions that specify reserved values of cntrl⟨2:0⟩ produce UNPREDICTABLE results. Cntrl⟨3⟩ should be zero; if it is set, the results of the instruction are UNPREDICTABLE.

VMULL Vector Integer Multiply

Format

```
opcode cntrl.rw                    ! Vector * Vector
opcode cntrl.rw, mulr.rl           ! Scalar * Vector
```

Vector Control Word

15	14	13	12	11 – 8	7 – 4	3 – 0
MOE	MTF	EXC	0	Va or 0	Vb	Vc

Operation

```
FOR i ← 0 TO VLR-1
   IF {{MOE EQL 0} OR
      {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
      BEGIN
      IF VVMULL THEN
         Vc[i]<31:0> ← Va[i]<31:0> * Vb[i]<31:0>
      IF VSMULL THEN
         Vc[i]<31:0> ← mulr * Vb[i]<31:0>
      END
```

VP Exceptions

Integer Overflow

Opcodes

A0FD	VVMULL	Vector Vector Multiply Longword
A1FD	VSMULL	Vector Scalar Multiply Longword

Description

The scalar multiplier or vector operand Va is multiplied, element-wise, by vector operand Vb, and the least significant 32 bits of the signed 64-bit product are written to vector register Vc. Only bits ⟨31:0⟩ of each vector element participate in the operation. Bits ⟨63:32⟩ of the elements of vector register Vc are UNPREDICTABLE. The length of the vector is specified by the VLR.

If integer overflow is detected and cntrl⟨EXC⟩ is set, the exception condition type and destination register number are recorded in the VAER, and the vector operation is allowed to complete. On integer overflow, the low-order 32 bits of the true result are stored in the destination element.

VSUBL Vector Integer Subtract

Format

```
opcode cntrl.rw                    ! Vector - Vector
opcode cntrl.rw, min.rl            ! Scalar - Vector
```

Vector Control Word

15	14	13	12	11 – 8	7 – 4	3 – 0
MOE	MTF	EXC	0	Va or 0	Vb	Vc

Operation

```
FOR i ← 0 TO VLR-1
   IF {{MOE EQL 0} OR
      {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
      BEGIN
      IF VVSUBL THEN
         Vc[i]<31:0> ← Va[i]<31:0> - Vb[i]<31:0>
      IF VSSUBL THEN
         Vc[i]<31:0> ← min - Vb[i]<31:0>
      END
```

VP Exceptions

Integer Overflow

Opcodes

88FD	VVSUBL	Vector Vector Subtract Longword
89FD	VSSUBL	Vector Scalar Subtract Longword

Description

The vector operand Vb is subtracted, element-wise, from the scalar minuend or vector operand Va. The 32-bit difference is written to vector register Vc. Only bits ⟨31:0⟩ of each vector element participate in the operation. Bits ⟨63:32⟩ of the elements of vector register Vc are UNPREDICTABLE. The length of the vector is specified by the VLR.

If integer overflow is detected and cntrl⟨EXC⟩ is set, the exception condition type and destination register number are recorded in the VAER, and the vector operation is allowed to complete. On integer overflow, the low-order 32 bits of the true result are stored in the destination element.

12.7 *Vector Logical and Shift Instructions*

This class of instructions performs logical and shift operations on vectors of longwords. The first source operand can be either an integer scalar value or a vector register containing longword data; the second source operand is always a vector register containing longword data.

VxlogL — Vector Logical Functions

Format

```
opcode cntrl.rw                 ! Vector op Vector
opcode cntrl.rw, src.rl         ! Vector op Scalar
```

Vector Control Word

15	14	13	12	11 — 8	7 — 4	3 — 0
MOE	MTF	0	0	Va or 0	Vb	Vc

Operation

```
FOR i ← 0 TO VLR-1
   IF {{MOE EQL 0} OR
      {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
      BEGIN
      IF VVBISL THEN
         Vc[i]<31:0> ← Va[i]<31:0> OR Vb[i]<31:0>
      IF VSBISL THEN
         Vc[i]<31:0> ← src OR Vb[i]<31:0>
      IF VVXORL THEN
         Vc[i]<31:0> ← Va[i]<31:0> XOR Vb[i]<31:0>
      IF VSXORL THEN
         Vc[i]<31:0> ← src XOR Vb[i]<31:0>
      IF VVBICL THEN
         Vc[i]<31:0> ← {NOT Va[i]<31:0>} AND Vb[i]<31:0>
      IF VSBICL THEN
         Vc[i]<31:0> ← {NOT src} AND Vb[i]<31:0>
      Vc[i]<63:32> ← Vb[i]<63:32>
      END
```

VP Exceptions

none

Opcodes

C8FD	VVBISL	Vector Vector Bit Set Longword
E8FD	VVXORL	Vector Vector Exclusive-OR Longword
CCFD	VVBICL	Vector Vector Bit Clear Longword
C9FD	VSBISL	Vector Scalar Bit Set Longword
E9FD	VSXORL	Vector Scalar Exclusive-OR Longword
CDFD	VSBICL	Vector Scalar Bit Clear Longword

Description

The scalar src or vector operand Va is combined, element-wise, using the specified boolean function, with vector register Vb; the result is written to vector register Vc. Only bits ⟨31:0⟩ of each

vector element participate in the operation. Bits ⟨63:32⟩ of the elements of Vb are written into bits ⟨63:32⟩ of the corresponding elements of Vc. The length of the vector is specified by the VLR.

VSL Vector Shift Logical

Format

```
opcode cntrl.rw                  ! Vector Shift Count
opcode cntrl.rw, cnt.rl          ! Scalar Shift Count
```

Vector Control Word

15	14	13	12	11 — 8	7 — 4	3 — 0
MOE	MTF	0	0	Va or 0	Vb	Vc

Operation

```
FOR i ← 0 TO VLR-1
   IF {{MOE EQL 0} OR
      {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
      BEGIN
      IF VVSRLL THEN
         Vc[i]<31:0> ← RIGHT_SHIFT(Vb[i]<31:0>, Va[i]<4:0>)
      IF VVSLLL THEN
         Vc[i]<31:0> ← LEFT_SHIFT(Vb[i]<31:0>,  Va[i]<4:0>)
      IF VSSRLL THEN
         Vc[i]<31:0> ← RIGHT_SHIFT(Vb[i]<31:0>, cnt<4:0>)
      IF VSSLLL THEN
         Vc[i]<31:0> ← LEFT_SHIFT(Vb[i]<31:0>,  cnt<4:0>)
      END
```

VP Exceptions

none

Opcodes

E0FD	VVSRLL	Vector Vector Shift Right Logical Longword
E4FD	VVSLLL	Vector Vector Shift Left Logical Longword
E1FD	VSSRLL	Vector Scalar Shift Right Logical Longword
E5FD	VSSLLL	Vector Scalar Shift Left Logical Longword

Description

Each element in vector register Vb is shifted logically left or right 0 to 31 bits as specified by a scalar count operand or vector register Va. The shifted results are written to vector register Vc. Zero bits are propagated into the vacated bit positions. Only bits ⟨4:0⟩ of the count operand and bits ⟨31:0⟩ of each Vb element participate in the operation. Bits ⟨63:32⟩ of the elements of vector register Vc are UNPREDICTABLE. The length of the vector is specified by the VLR.

12.8 Vector Floating-Point Instructions

The VAX vector architecture provides instructions for operating on F_floating, D_floating, and G_floating operand formats. The floating-point arithmetic instructions are add, subtract, compare, multiply, and divide. Data conversion instructions are provided to convert operands between D_floating, G_floating, F_floating, and longword integer.

Rounding is performed using standard VAX rounding rules. The accuracy of the vector floating-point instructions matches that of the scalar floating-point instructions. Refer to section 3.10.

All vector floating-point exception conditions occur asynchronously with respect to the scalar processor. These exception conditions do not interrupt the scalar processor. If the exception condition is enabled, the exception condition type and destination register number are recorded in the vector arithmetic exception register, and a reserved operand in the format of the instruction's data type is written into the destination register element. Encoded in this reserved operand is the exception condition type. After recording the exception and writing the appropriate result into the destination register element, the instruction encountering the exception continues executing to completion.

If a vector convert-floating-to-integer instruction encounters a source element that is a reserved operand, an UNPREDICTABLE result rather than a reserved operand is written into the destination register element.

Figure 12.10 shows the encoding of the reserved operand that is written for vector floating-point exceptions. Consistent with the definition of a reserved operand (as defined in section 3.10), the sign bit (bit ⟨15⟩) is 1 and the exponent (bits ⟨14:7⟩ for F_floating and D_floating, and bits ⟨14:4⟩ for G_floating) is 0. When the reserved operand is written in F_floating or D_floating format, bits ⟨6:4⟩ are also 0. The exception condition type (ETYPE) is encoded in bits ⟨3:0⟩, as shown in Table 12.7. If a reserved operand is divided by zero, both ETYPE bits may be set. The state of all other bits in the result (denoted with an "x") is UNPREDICTABLE.

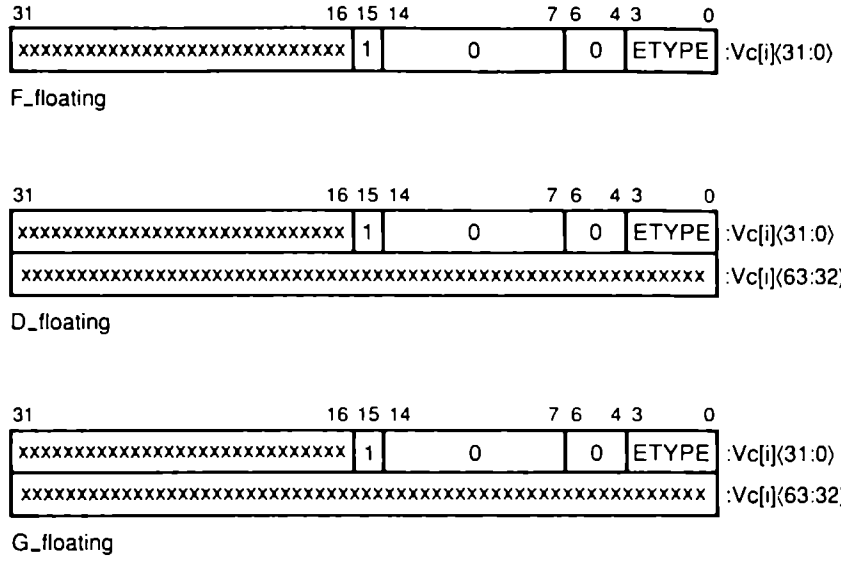


Figure 12.10
Encoding of the Reserved Operand

Table 12.7
Encoding of the Exception Condition Type (ETYPE)

Bit	Exception Condition Type
⟨0⟩	Floating Underflow
⟨1⟩	Floating Divide by Zero
⟨2⟩	Floating Reserved Operand
⟨3⟩	Floating Overflow

If the floating underflow exception condition is suppressed by cntrl⟨EXC⟩, a zero result is written to the destination register element, and no further action is taken. Floating overflow, floating divide by zero, and floating reserved operand are always enabled.

VADD — Vector Floating Add

Format

```
opcode cntrl.rw                  ! Vector + Vector
opcode cntrl.rw, addend.rl       ! Scalar + Vector (F)
opcode cntrl.rw, addend.rq       ! Scalar + Vector (D and G)
```

Vector Control Word

15	14	13	12	11 — 8	7 — 4	3 — 0
MOE	MTF	EXC	0	Va or 0	Vb	Vc

Operation

```
FOR i ← 0 TO VLR-1
  IF {{MOE EQL 0} OR {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
     BEGIN
     IF VVADDF THEN
        Vc[i]<31:0> ← Va[i]<31:0> + Vb[i]<31:0>
     IF VSADDF THEN
        Vc[i]<31:0> ← addend + Vb[i]<31:0>
     IF VVADDD OR VVADDG THEN
        Vc[i] ← Va[i] + Vb[i]
     IF VSADDD OR VSADDG THEN
        Vc[i] ← addend + Vb[i]
     END
```

VP Exceptions

```
Floating Overflow
Floating Reserved Operand
Floating Underflow
```

Opcodes

84FD	VVADDF	Vector Vector Add F_Floating
85FD	VSADDF	Vector Scalar Add F_Floating
86FD	VVADDD	Vector Vector Add D_Floating
87FD	VSADDD	Vector Scalar Add D_Floating
82FD	VVADDG	Vector Vector Add G_Floating
83FD	VSADDG	Vector Scalar Add G_Floating

Description

The source addend or vector operand Va is added, element-wise, to vector register Vb, and the sum is written to vector register Vc. The length of the vector is specified by the VLR.

In VxADDF, only bits ⟨31:0⟩ of each vector element participate in the operation. Bits ⟨63:32⟩ of the destination vector elements are UNPREDICTABLE.

If a floating underflow occurs when cntrl⟨EXC⟩ is set or if a floating overflow or floating reserved operand occurs, an encoded reserved operand is stored as the result, and the exception condition type and destination register number are recorded in the VAER. The vector operation is then allowed to complete. If floating underflow is detected when cntrl⟨EXC⟩ is clear, zero is written to the destination element and no other action is taken.

VCMP Vector Floating Compare

Format

```
opcode cntrl.rw                  ! Vector - Vector
opcode cntrl.rw, src.rl          ! Scalar - Vector (F)
opcode cntrl.rw, src.rq          ! Scalar - Vector (D and G)
```

Vector Control Word

15	14	13	12	11 – 8	7 – 4	3 – 0
MOE	MTF	0	0	Va or 0	Vb	cmp func

Operation

```
CASE cntrl<3:0> OF
   0:          SIGNED_RELATION is Greater Than;
   1:          SIGNED_RELATION is Equal;
   2:          SIGNED_RELATION is Less Than;
   4:          SIGNED_RELATION is Less Than or Equal;
   5:          SIGNED_RELATION is Not Equal;
   6:          SIGNED_RELATION is Greater Than or Equal;
   otherwise: SIGNED_RELATION is Reserved;
   END

FOR i ← 0 TO VLR-1
   IF {{MOE EQL 0} OR
      {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
      BEGIN
      IF VVCMPF THEN
         IF Va[i]<31:0> SIGNED_RELATION Vb[i]<31:0> THEN
            VMR<i> ← 1
         ELSE
            VMR<i> ← 0
      IF VVCMPD OR VVCMPG THEN
         IF Va[i] SIGNED_RELATION Vb[i] THEN
            VMR<i> ← 1
         ELSE
            VMR<i> ← 0
      IF VSCMPF THEN
         IF src SIGNED_RELATION Vb[i]<31:0> THEN
            VMR<i> ← 1
         ELSE
            VMR<i> ← 0
      IF VSCMPD OR VSCMPG THEN
         IF src SIGNED_RELATION Vb[i] THEN
            VMR<i> ← 1
         ELSE
            VMR<i> ← 0
      END
```

VP Exceptions Floating Reserved Operand

Opcodes

C4FD	VVCMPF	Vector Vector Compare F_floating
C5FD	VSCMPF	Vector Scalar Compare F_floating
C6FD	VVCMPD	Vector Vector Compare D_floating
C7FD	VSCMPD	Vector Scalar Compare D_floating
C2FD	VVCMPG	Vector Vector Compare G_floating
C3FD	VSCMPG	Vector Scalar Compare G_floating

Description

The scalar or vector operand Va is compared, element-wise, with vector register Vb. The length of the vector is specified by the VLR. For each element comparison, if the specified relationship is true, the vector mask register bit (VMR⟨i⟩) corresponding to the vector element is set to 1; otherwise it is cleared. If cntrl⟨MOE⟩ is set, the VMR bits corresponding to elements that do not match cntrl⟨MTF⟩ are left unchanged. VMR bits beyond the vector length are left unchanged. If an element being compared is a reserved operand, VMR⟨i⟩ is UNPREDICTABLE. In VxCMPF, only bits ⟨31:0⟩ of each vector element participate in the operation.

If a floating reserved operand exception occurs, the exception condition type is recorded in the VAER and the vector operation is allowed to complete.

Note that for this instruction, no bits are set in the VAER destination register mask when an exception occurs.

Vector floating compare instructions that specify reserved values of cntrl⟨2:0⟩ produce UNPREDICTABLE results. Cntrl⟨3⟩ should be 0; if it is set, the results of the instruction are UNPREDICTABLE.

VVCVT Vector Convert

Format `opcode cntrl.rw`

Vector Control Word

15	14	13	12	11 — 8	7 — 4	3 — 0
MOE	MTF	EXC	0	cvt func	Vb	Vc

The convert operation is determined by cntrl⟨11:8⟩, as shown below:

cntrl⟨11:8⟩	Name	Interpretation
1 1 1 1	CVTRGL	Convert Rounded G_Floating to Longword
1 1 1 0	Reserved	
1 1 0 1	CVTGF	Convert Rounded G_Floating to F_Floating
1 1 0 0	CVTGL	Convert Truncated G_Floating to Longword
1 0 1 1	Reserved	
1 0 1 0	CVTRDL	Convert Rounded D_Floating to Longword
1 0 0 1	CVTDF	Convert Rounded D_Floating to F_Floating
1 0 0 0	CVTDL	Convert Truncated D_Floating to Longword
0 1 1 1	CVTFG	Convert F_Floating to G_Floating (exact)
0 1 1 0	CVTFD	Convert F_Floating to D_Floating (exact)
0 1 0 1	CVTRFL	Convert Rounded F_Floating to Longword
0 1 0 0	CVTFL	Convert Truncated F_Floating to Longword
0 0 1 1	CVTLG	Convert Longword to G_Floating (exact)
0 0 1 0	CVTLD	Convert Longword to D_Floating (exact)
0 0 0 1	CVTLF	Convert Rounded Longword to F_Floating
0 0 0 0	Reserved	

Vector convert instructions produce UNPREDICTABLE results if they specify reserved values of cntrl⟨11:8⟩.

Operation

```
FOR i ← 0 TO VLR-1
   IF {{MOE EQL 0} OR
      {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
      Vc[i] ← {conversion of Vb[i] based on cntrl<11:8>}
```

VP Exceptions

```
Floating Overflow
Floating Reserved Operand
Floating Underflow
Integer Overflow
```

Opcodes ECFD VVCVT Vector Convert

Description The vector elements in vector register Vb are converted, and results are written to vector register Vc. Cntrl⟨11:8⟩ specifies the conversion to be performed. The length of the vector is specified

by the VLR. Bits ⟨63:32⟩ of Vc are UNPREDICTABLE for instructions that convert from D_floating or G_floating to F_floating or longword. When CVTRGL, CVTRDL, and CVTRFL round, the rounding is done in sign magnitude, before conversion to two's complement.

If an integer overflow occurs when cntrl⟨EXC⟩ is set, the low-order 32 bits of the true result are written to the destination element as the result, and the exception condition type and destination register number are recorded in the VAER. The vector operation is then allowed to complete. If integer overflow occurs when cntrl⟨EXC⟩ is clear, the low-order 32 bits of the true result are written to the destination element, and no other action is taken.

Vector convert-floating-to-integer instructions, where the source element is a reserved operand, result in a destination element that is UNPREDICTABLE. In addition, the exception type and destination register number are recorded in the VAER. The vector operation is then allowed to complete.

For vector convert-floating-to-floating instructions, if floating underflow occurs when cntrl⟨EXC⟩ is clear, zero is written to the destination element, and no other action is taken. The vector operation is then allowed to complete.

For vector convert-floating-to-floating instructions, if floating underflow occurs with cntrl⟨EXC⟩ set or if a floating overflow or reserved operand occurs, an encoded reserved operand is written to the destination element, and the exception condition type and destination register number are recorded in the VAER. The vector operation is then allowed to complete.

VDIV Vector Floating Divide

Format

```
opcode cntrl.rw                 ! Vector / Vector
opcode cntrl.rw, divd.rl        ! Scalar / Vector (F)
opcode cntrl.rw, divd.rq        ! Scalar / Vector (D and G)
```

Vector Control Word

15	14	13	12	11 — 8	7 — 4	3 — 0
MOE	MTF	EXC	0	Va or 0	Vb	Vc

Operation

```
FOR i ← 0 TO VLR-1
   IF {{MOE EQL 0} OR
      {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
      BEGIN
      IF VVDIVF THEN
         Vc[i]<31:0> ← Va[i]<31:0> / Vb[i]<31:0>
      IF VSDIVF THEN
         Vc[i]<31:0> ← divd / Vb[i]<31:0>
      IF VVDIVD OR VVDIVG THEN
         Vc[i] ← Va[i] / Vb[i]
      IF VSDIVD OR VSDIVG THEN
         Vc[i] ← divd / Vb[i]
      END
```

VP Exceptions

```
Floating Divide by Zero
Floating Overflow
Floating Reserved Operand
Floating Underflow
```

Opcodes

ACFD	VVDIVF	Vector Vector Divide F_floating
ADFD	VSDIVF	Vector Scalar Divide F_floating
AEFD	VVDIVD	Vector Vector Divide D_floating
AFFD	VSDIVD	Vector Scalar Divide D_floating
AAFD	VVDIVG	Vector Vector Divide G_floating
ABFD	VSDIVG	Vector Scalar Divide G_floating

Description

The scalar dividend or vector register Va is divided, element-wise, by the divisor in vector register Vb, and the quotient is written to vector register Vc. The length of the vector is specified by the VLR.

In VxDIVF, only bits ⟨31:0⟩ of each vector element participate in the operation; bits ⟨63:32⟩ of the destination vector elements are UNPREDICTABLE.

If a floating underflow occurs when cntrl⟨EXC⟩ is set or if a floating overflow, divide by zero, or reserved operand occurs, an encoded reserved operand is stored as the result, and the exception

condition type and destination register number are recorded in the VAER. The vector operation is then allowed to complete. If floating underflow is detected when cntrl⟨EXC⟩ is clear, zero is written to the destination element and no other action is taken.

VMUL Vector Floating Multiply

Format

```
opcode cntrl.rw                 ! Vector * Vector
opcode cntrl.rw, mulr.rl        ! Scalar * Vector (F)
opcode cntrl.rw, mulr.rq        ! Scalar * Vector (D and G)
```

Vector Control Word

15	14	13	12	11 – 8	7 – 4	3 – 0
MOE	MTF	EXC	0	Va or 0	Vb	Vc

Operation

```
FOR i ← 0 TO VLR-1
   IF {{MOE EQL 0} OR
       {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
      BEGIN
      IF VVMULF THEN
         Vc[i]<31:0> ← Va[i]<31:0> * Vb[i]<31:0>
      IF VSMULF THEN
         Vc[i]<31:0> ← mulr * Vb[i]<31:0>
      IF VVMULD OR VVMULG THEN
         Vc[i] ← Va[i] * Vb[i]
      IF VSMULD OR VSMULG THEN
         Vc[i] ← mulr * Vb[i]
      END
```

VP Exceptions

```
Floating Overflow
Floating Reserved Operand
Floating Underflow
```

Opcodes

A4FD	VVMULF	Vector Vector Multiply F_floating
A5FD	VSMULF	Vector Scalar Multiply F_floating
A6FD	VVMULD	Vector Vector Multiply F_floating
A7FD	VSMULD	Vector Scalar Multiply D_floating
A2FD	VVMULG	Vector Vector Multiply G_floating
A3FD	VSMULG	Vector Scalar Multiply G_floating

Description

The multiplicand in vector register Vb is multiplied, element-wise, by the scalar multiplier or vector operand Va, and the product is written to vector register Vc. The length of the vector is specified by the VLR.

In VxMULF, only bits ⟨31:0⟩ of each vector element participate in the operation. Bits ⟨63:32⟩ of the destination vector elements are UNPREDICTABLE.

If a floating underflow occurs when cntrl⟨EXC⟩ is set or if a floating overflow or reserved operand occurs, an encoded reserved operand is stored as the result, and the exception condition type and

destination register number are recorded in the VAER. The vector operation is then allowed to complete. If floating underflow is detected when cntrl⟨EXC⟩ is clear, zero is written to the destination element and no other action is taken.

VSUB Vector Floating Subtract

Format

```
opcode cntrl.rw                 ! Vector - Vector
opcode cntrl.rw, min.rl         ! Scalar - Vector (F)
opcode cntrl.rw, min.rq         ! Scalar - Vector (D and G)
```

Vector Control Word

15	14	13	12	11 – 8	7 – 4	3 – 0
MOE	MTF	EXC	0	Va or 0	Vb	Vc

Operation

```
FOR i ← 0 TO VLR-1
   IF {{MOE EQL 0} OR
      {{MOE EQL 1} AND {VMR<i> EQL MTF}}} THEN
      BEGIN
      IF VVSUBF THEN
         Vc[i]<31:0> ← Va[i]<31:0> - Vb[i]<31:0>
      IF VSSUBF THEN
         Vc[i]<31:0> ← min - Vb[i]<31:0>
      IF VVSUBD OR VVSUBG THEN
         Vc[i] ← Va[i] - Vb[i]
      IF VSSUBD OR VSSUBG THEN
         Vc[i] ← min - Vb[i]
      END
```

VP Exceptions

```
Floating Overflow
Floating Reserved Operand
Floating Underflow
```

Opcodes

8CFD	VVSUBF	Vector Vector Subtract F_floating
8DFD	VSSUBF	Vector Scalar Subtract F_floating
8EFD	VVSUBD	Vector Vector Subtract D_floating
8FFD	VSSUBD	Vector Scalar Subtract D_floating
8AFD	VVSUBG	Vector Vector Subtract G_floating
8BFD	VSSUBG	Vector Scalar Subtract G_floating

Description

Vector register Vb is subtracted, element-wise, from the scalar minuend or vector register Va, and the difference is written to vector register Vc. The length of the vector is specified by the VLR.

In VxSUBF, only bits ⟨31:0⟩ of each vector element participate in the operation; bits ⟨63:32⟩ of the destination vector elements are UNPREDICTABLE.

If a floating underflow occurs when cntrl⟨EXC⟩ is set or if a floating overflow or reserved operand occurs, an encoded reserved operand is stored as the result, and the exception condition type and

destination register number are recorded in the VAER. The vector operation is then allowed to complete. If floating underflow is detected when cntrl⟨EXC⟩ is clear, zero is written to the destination element and no other action is taken.

12.9 Vector Edit Instructions

This class of instructions performs edit operations on vector registers. Operations such as merging two vector registers into a third vector register or placing an iterated offset vector (an IOTA vector) into a vector register are examples.

VMERGE Vector Merge

Format

```
opcode cntrl.rw                 ! Vector Vector Merge
opcode cntrl.rw, src.rq         ! Vector Scalar Merge
```

Vector Control Word

15	14	13	12	11 — 8	7 — 4	3 — 0
0	MTF	0	0	Va or 0	Vb	Vc

Operation

```
FOR i ← 0 TO VLR-1
   BEGIN
   IF VVMERGE THEN
      IF {VMR<i> EQL MTF} THEN
         Vc[i] ← Va[i]
      ELSE
         Vc[i] ← Vb[i]
   IF VSMERGE THEN
      IF {VMR<i> EQL MTF} THEN
         Vc[i] ← src
      ELSE
         Vc[i] ← Vb[i]
   END
```

VP Exceptions none

Opcodes

EEFD	VVMERGE	Vector Vector Merge
EFFD	VSMERGE	Vector Scalar Merge

Description The scalar src or vector operand Va is merged, element-wise, with vector register Vb, and the resulting vector is written to vector register Vc. The length of the vector operation is specified by the VLR.

For each vector element, i, if the corresponding vector mask register bit (VMR⟨i⟩) matches cntrl⟨MTF⟩, src or Va[i] is written to the destination vector element Vc[i]. If VMR⟨i⟩ does not match cntrl⟨MTF⟩, Vb[i] is written to the destination vector element.

IOTA — Generate Compressed Iota Vector

Format

```
opcode cntrl.rw, stride.rl
```

Vector Control Word

15	14	13	12	11 – 8	7 – 4	3 – 0
0	MTF	0	0	0	0	Vc

Operation

```
j   ← 0
tmp ← 0
FOR i ← 0 TO VLR-1
   BEGIN
   IF {VMR<i> EQL MTF} THEN
      BEGIN
      Vc[j]<31:0> ← tmp<31:0>
      j ← j + 1
      END
   tmp ← tmp + stride
   END
VCR ← j   !return vector count
```

VP Exceptions

none

Opcodes

EDFD IOTA Generate Compressed Iota Vector

Description

IOTA constructs a vector of offsets for use by the vector gather and scatter instructions VGATH and VSCAT.

IOTA first generates an iota vector of length VLR using the stride operand. An iota vector is a vector whose first element is zero and whose subsequent elements are spaced by the stride increment. The stride can be positive, negative, or zero. For example,

```
0*stride, 1*stride, 2*stride, 3*stride, ..., VLR-1*stride
```

The iota vector is then compressed using the contents of the VMR. Elements of the iota vector for which the corresponding VMR bit matches cntrl⟨MTF⟩ are written in contiguous elements of the destination vector register Vc. Only bits ⟨31:0⟩ of each iota and destination vector element participate in the operation. Bits ⟨63:32⟩ of the destination vector elements are UNPREDICTABLE.

The number of elements written to Vc is returned in the vector count register (VCR). The values of elements in the destination vector register between the new value of the VCR and the vector length are UNPREDICTABLE.

Note

If a large value is specified for the stride.rl operand, there is a chance for integer overflow during calculation of the "tmp ← tmp + stride" step. In this case, the overflow is ignored. For example:

```
tmp ← tmp + stride

Value of tmp before this step is:     FFFFFF00
Value of Stride is:                   FFFFFF00

Value of tmp + stride is:           1 FFFFFE00

Since the overflow is ignored,
the new value of tmp is:              FFFFFE00
```

12.10 Miscellaneous Instructions

This class of instructions performs miscellaneous, user mode, control functions on the vector processor. Functions such as reading or writing a vector control register, providing scalar/vector memory synchronization, or ensuring vector exception-reporting synchronization are examples.

MFVP — Move From Vector Processor

Format

```
opcode regnum.rw, dst.wl
```

Operation

```
CASE regnum OF
   0:   dst ← ZEXT{VCR}
   1:   dst ← ZEXT{VLR}
   2:   dst ← VMR<31:0>
   3:   dst ← VMR<63:32>
   4:   SYNC
        dst ← UNPREDICTABLE
   5:   MSYNC
        dst ← UNPREDICTABLE
   >5:  Reserved
END
```

VP Exceptions

none

Opcodes

31FD MFVP Move From Vector Processor

Description

This instruction can be used to read the vector count, length, and mask registers and to synchronize a scalar processor with its associated vector processor.

When the scalar processor issues an MFVP instruction to the vector processor, the scalar processor waits for the MFVP result to be written before processing other instructions.

Move From Vector Processor instructions that specify reserved values of the regnum operand produce UNPREDICTABLE results.

MFVP from the VCR or the VLR does not read that register until all previous writes to the register are completed. MFVP from VMR⟨31:0⟩ or VMR⟨63:32⟩ does not read that longword of the VMR until all previous writes to the same longword of the VMR are completed; however, this is not true for previous writes to the other longword.

SYNC allows software to ensure that the unreported exceptions of all previously issued vector instructions (including vector memory instructions in asynchronous memory management mode) are detected and reported to the scalar processor before the scalar processor proceeds with further instructions. (For more details about SYNC and its exception reporting nature refer to section 13.3.1.)

MSYNC allows software to ensure that all previously issued memory instructions of the scalar/vector processor pair are complete before the scalar processor proceeds with further instructions. (For more details about MSYNC and its exception reporting nature, refer to section 13.3.2.)

The value of the vector control register (VCR, VLR, VMR⟨31:0⟩, VMR⟨63:32⟩) delivered by an MFVP depends on the value of certain vector register elements and vector control register bits. Unreported exceptions that occur in the production of these elements and control register bits are reported by the vector processor prior to the completion of the MFVP from the vector control register.

In addition, there are vector register elements and vector control register bits on which the value of a vector control register delivered by an MFVP does not depend on. It is UNPREDICTABLE whether unreported exceptions that occur in the production of these elements and control register bits are reported by the vector processor prior to the completion of the MFVP from the vector control register. Software must not rely on the reporting of these exceptions prior to the completion of the MFVP for the correctness of program results.

Section 13.1.3 lists the rules necessary to determine on which vector control register elements and vector control register bits the value of a vector control register delivered by an MFVP depends. Examples of MFVP exception reporting using these rules are in section 13.2.5.

When a vector arithmetic exception or memory management exception (in asynchronous memory management mode) is reported prior to the completion of an MFVP, the following actions occur:

1. the operation of the MFVP does not complete,
2. no longword result is written to the scalar destination of the MFVP by the scalar processor, and
3. the MFVP itself (rather than the next vector instruction) takes either a vector processor disabled fault or a memory management fault.

After the appropriate fault has been serviced, the MFVP may be returned to through an REI. If both exception conditions are encountered by an MFVP, then the MFVP itself takes a vector processor disabled fault. In this case, after the vector processor disabled fault has been serviced, returning to the MFVP instruction will cause the asynchronous memory management exception to be reported.

MTVP Move To Vector Processor

Format

```
opcode regnum.rw, src.rl
```

Operation

```
CASE regnum OF
   0:   VCR ← src
   1:   VLR ← src
   2:   VMR<31:0> ← src
   3:   VMR<63:32> ← src
   >3:  Reserved
END
```

VP Exceptions

none

Opcodes

A9FD MTVP Move To Vector Processor

Description

This instruction can be used to write the vector count, length, and mask registers.

The new value of the VCR, VLR, or VMR does not affect any prior instructions. The new value remains in effect for all subsequent vector instructions executed until a new value is loaded.

Move To Vector Processor instructions that specify reserved values of the regnum operand produce UNPREDICTABLE results.

VSYNC — Synchronize Vector Memory Access

Format

```
opcode regnum.rw
```

Operation

```
CASE regnum OF
   0:   VSYNC
   >0:  Reserved
END
```

VP Exceptions

none

Opcodes

A8FD VSYNC Synchronize Vector Memory Access

Description

The VSYNC instruction can be used to synchronize memory access within the vector processor. The instruction allows software to order the conflicting memory accesses of vector-memory instructions issued after VSYNC with those of vector-memory instructions issued before VSYNC. Specifically, VSYNC forces the access of a memory location by any subsequent vector-memory instruction to wait for (depend on) the completion of all prior conflicting accesses of that location by previous vector-memory instructions. (See section 13.3.4 for more details.)

See section 13.3.5 for the conditions when VSYNC is not required before a vector store instruction.

Notes

1. Synchronize Vector Memory Access instructions that specify reserved values of the regnum operand produce UNPREDICTABLE results.

12.11 Assembler Notation

The assembler notation uses a format that is different from the operand specifiers for the vector instructions. The number and order of operands are not the same as the instruction-stream format. For example, vector-to-vector addition is denoted by the assembler as *VVADDL V1, V2, V3* instead of *VVADDL ^X123*. The assembler always generates immediate addressing mode (I^#constant) for vector control word operands.

The assembler notation for vector instructions uses opcode qualifiers to select whether vector processor exception conditions are enabled or disabled and to select the value of cntrl⟨MTF⟩ in masked, VMERGE, and IOTA operations. The appropriate opcode is followed by a slash (/) and up to two of the following qualifiers specified in any order.

- The qualifier U enables floating underflow. The qualifier V enables integer overflow. Both of these qualifiers set cntrl⟨EXC⟩. The default is no vector processor exception conditions are enabled.
- The qualifier 0 denotes masked operation on elements for which the vector mask register (VMR) bit is 0. The qualifier 1 denotes masked operation on elements for which the VMR bit is 1. Both qualifiers set cntrl⟨MOE⟩. The default is no masked operations.
- For VMERGE and IOTA instructions only, the qualifier 0 denotes cntrl⟨MTF⟩ is 0. The qualifier 1 denotes cntrl⟨MTF⟩ is 1. Cntrl⟨MTF⟩ is 1 by default. Cntrl⟨MOE⟩ is not set in this case.
- For VLD and VGATH instructions only, the qualifier M indicates modify intent (cntrl⟨MI⟩ is 1). The default is no modify intent (cntrl⟨MI⟩ is 0).

Examples

```
VVADDF/1    V0, V1, V2    ;Operates on elements with
                          ; mask bit set
VVMULD/0    V0, V1, V2    ;Operates on elements with
                          ; mask bit clear
VVADDL/V    V0, V1, V2    ;Enables exception conditions
                          ; (integer overflow here)
```

Table 12.8
Assembler Notation for Vector Instructions (Part 1)

Assembler Notation		Vector Instruction Assembled	
VLDx	base, stride, Vc	VLDx	cntrl.rw, base.ab, stride.rl
VGATHx	base, Vb, Vc	VGATHx	cntrl.rw, base.ab
VSTx	Vc, base, stride	VSTx	cntrl.rw, base.ab, stride.rl
VSCATx	Vc, base, Vb	VSCATx	cntrl.rw, base.ab
		(where x = {L,Q})	
VVADDx	Va, Vb, Vc	VVADDx	cntrl.rw
VSADDx	scalar, Vb, Vc	VSADDx	cntrl.rw, addend.ry
VVSUBx	Va, Vb, Vc	VVSUBx	cntrl.rw
VSSUBx	scalar, Vb, Vc	VSSUBx	cntrl.rw, min.ry
VVMULx	Va, Vb, Vc	VVMULx	cntrl.rw
VSMULx	scalar, Vb, Vc	VSMULx	cntrl.rw, mulr.ry
VVGTRx	Va, Vb	VVCMPx	cntrl.rw {cmp_func = GTR}
VSGTRx	src, Vb	VSCMPx	cntrl.rw, src.ry {cmp_func = GTR}
VVEQLx	Va, Vb	VVCMPx	cntrl.rw {cmp_func = EQL}
VSEQLx	src, Vb	VSCMPx	cntrl.rw, src.ry {cmp_func = EQL}
VVLSSx	Va, Vb	VVCMPx	cntrl.rw {cmp_func = LSS}
VSLSSx	src, Vb	VSCMPx	cntrl.rw, src.ry {cmp_func = LSS}
VVNEQx	Va, Vb	VVCMPx	cntrl.rw {cmp_func = NEQ}
VSNEQx	src, Vb	VSCMPx	cntrl.rw, src.ry {cmp_func = NEQ}
VVGEQx	Va, Vb	VVCMPx	cntrl.rw {cmp_func = GEQ}
VSGEQx	src, Vb	VSCMPx	cntrl.rw, src.ry {cmp_func = GEQ}
VVLEQx	Va, Vb	VVCMPx	cntrl.rw {cmp_func = LEQ}
VSLEQx	src, Vb	VSCMPx	cntrl.rw, src.ry {cmp_func = LEQ}
		(where x = {L,F} and y = {L}, or x = {D,G} and y = {Q})	
VVDIVx	Va, Vb, Vc	VVDIVx	cntrl.rw
VSDIVx	scalar, Vb, Vc	VSDIVx	cntrl.rw, divd.ry
		(where x = {F} and y = {L}, or x = {D,G} and y = {Q})	

```
VVSUBG/UO   V0, V1, V2    ;Enables floating underflow and
                          ; operates on elements with
                          ; mask bit clear
VLDL/M      base,#4,V1    ;Indicates modify intent
```

The assembler notation is shown in in Table 12.8.

Table 12.8
Assembler Notation for Vector Instructions (Part 2)

Assembler Notation	Vector Instruction Assembled
VVCVTxy Vb, Vc	VVCVT cntrl.rw {cvt_func = xy} (where xy = {FD, FG, GF, DF, LF,LD,LG, FL,DL,GL, RFL,RDL,RGL})
VVBICL Va, Vb, Vc	VVBICL cntrl.rw
VSBICL scalar, Vb, Vc	VSBICL cntrl.rw, src.rl
VVBISL Va, Vb, Vc	VVBISL cntrl.rw
VSBISL scalar, Vb, Vc	VSBISL cntrl.rw, src.rl
VVXORL Va, Vb, Vc	VVXORL cntrl.rw
VSXORL scalar, Vb, Vc	VSXORL cntrl.rw, src.rl
VVSLLL Va, Vb, Vc	VVSLLL cntrl.rw
VSSLLL cnt, Vb, Vc	VSSLLL cntrl.rw, cnt.rl
VVSRLL Va, Vb, Vc	VVSRLL cntrl.rw
VSSRLL cnt, Vb, Vc	VSSRLL cntrl.rw, cnt.rl
IOTA stride, Vc	IOTA cntrl.rw, stride.rl
VVMERGE Va, Vb, Vc	VVMERGE cntrl.rw
VSMERGE src, Vb, Vc	VSMERGE cntrl.rw, src.rq
VSMERGEx src, Vb, Vc	VSMERGE cntrl.rw, src.rq[a]
MTVCR src	MTVP S^#0, src.wl
MFVCR dst	MFVP S^#0, dst.wl
MTVLR src	MTVP S^#1, src.wl
MFVLR dst	MFVP S^#1, dst.wl
MTVMRLO src	MTVP S^#2, src.wl
MFVMRLO dst	MFVP S^#2, dst.wl
MTVMRHI src	MTVP S^#3, src.wl
MFVMRHI dst	MFVP S^#3, dst.wl
SYNC dst	MFVP S^#4, dst.wl
MSYNC dst	MFVP S^#5, dst.wl
VSYNC	VSYNC S^#0

[a]For VSMERGEx, x = (F,D,G) and determines the floating-point format the assembler will use when src is immediate to encode src as the quadword-length scalar source operand. VSMERGEF creates a scalar source operand whose high-order longword is all zeroes and whose low-order longword is the F_floating point encoding of src. Literal mode is not allowed with the VSMERGEx pseudo-opcodes; attempting to force use of this address mode will result in an assembly warning. Likewise, non-immediate mode addressing is not allowed with VSMERGEF, and attempting to force use of this address mode will result in an assembly warning. When src is not immediate or literal, VSMERGEG and VSMERGED assemble exactly like VSMERGE.

13

VAX Vector Execution Model and Exception Reporting

13.1 Execution Model

A typical processor consists of a VAX scalar processor and its associated vector processor, which contains vector registers and vector function units. The scalar and vector processors may execute asynchronously. The VAX scalar processor decodes both scalar and vector instructions following the operand specifier evaluation rules stated in Chapter 2, but executes only the scalar instructions. The scalar processor passes the information required to execute a vector instruction to the vector processor. This information may include the vector opcode, scalar source operands, and vector control words. The vector processor performs the required operation, such as loading data from memory, storing data to memory, or manipulating data already loaded into its vector registers.

The scalar processor may decode a vector instruction before it determines whether the vector processor should receive it. Exceptions on vector instruction operands may occur during this decoding and be taken before the attempt to send the decoded instruction to the vector processor. The scalar processor follows the operation shown below when sending a decoded vector instruction to the vector processor. Remember that because the vector and scalar processors can execute asynchronously, a VPSR state transition may not be seen immediately by the scalar processor.

- If the scalar processor views the vector processor as enabled (the scalar processor sees VPSR⟨VEN⟩ as set), the decoded

vector instruction is sent to the vector processor. The vector processor queues instructions sent by the scalar processor until they can be executed.

- If the scalar processor views the vector processor as disabled (the scalar processor sees VPSR⟨VEN⟩ as clear), attempting to send the decoded vector instruction to the vector processor results in a vector processor disabled fault.

The following pseudocode details how vector instruction decode proceeds from the scalar processor:

```
DO WHILE (the scalar processor has a decoded vector
          instruction for the vector processor)
   BEGIN
   IF (the scalar processor sees VPSR<VEN> as clear) THEN
      enter the vector processor disabled fault handler.
   ELSE
      IF (asynchronous memory management handling is
         implemented AND VPSR<PMF> is set) THEN
      enter the memory management exception handler.
      The vector processor clears VPSR<PMF>.
   ELSE
      BEGIN
      If asynchronous memory management handling is
      implemented and VPSR<MF> is set, the vector processor
      clears VPSR<MF>, and retries the faulting memory
      reference before any new vector instructions in the
      queue are executed.
      IF (vector processor instruction queue not full) THEN
         BEGIN
         Send the decoded instruction to the vector processor
         for execution.
         IF (the decoded instruction is a vector memory
             access instruction AND synchronous memory
             management handling is implemented) THEN
            ensure instruction completion without the
            occurrence of memory management exceptions.
         END
      END
END
```

If asynchronous memory management handling is implemented and VPSR⟨MF⟩ is set when the scalar processor sends the vector processor an instruction, the vector processor clears VPSR⟨MF⟩ and retries the faulting memory reference before any new vector instructions in the queue are executed.

The VAX scalar processor need not wait for the vector processor to complete its operation before processing other instructions. Thus, the scalar processor could be processing other VAX instructions while the vector processor is performing vector operations. However, if the scalar processor issues an MFVP instruction to the vector processor, the scalar processor must wait for the MFVP result to be written before processing other instructions.

Because the scalar and vector processors may execute asynchronously, it is possible to perform a context switch in the scalar processor before the vector processor is idle. Software is responsible for ensuring that scalar and vector memory management remains synchronized and that all exceptions are reported in the context of the process where they occurred. This is achieved by making sure all vector memory accesses complete and then disabling the vector processor before any scalar context switch.

The vector processor may have its own TB and cache and may have separate paths to memory, or it may share these resources with the scalar processor.

13.1.1 *Access Mode Restrictions*

In general, processes are expected to use the vector processor in only one mode. However, multi-mode use of the vector processor by a process is allowed. Software decides whether to allow vector processor exceptions from vector instructions executed in a previous access mode to be reported in the current mode. The preferred method is to report all vector processor exceptions in the access mode where they occurred. This is achieved by requiring a process that uses the vector processor to execute a SYNC instruction before changing to an access mode where additional vector instructions are executed. The alternative is for software to handle vector processor exceptions resulting from vector instructions executed in a previous mode.

For correct access checking of vector memory references, the vector processor must know the access mode in effect when a vector memory access instruction is issued by the scalar processor.

13.1.2 *Scalar Context Switching*

With the addition of a vector processor, the steps required to per-

form a scalar context switch change. The following outlines the required method software should use for scalar context switching:

1. Disable the vector processor so that no new vector instructions will be accepted. Writing zero to the VPSR using the MTPR instruction clears VPSR⟨VEN⟩ and disables the vector processor without affecting VPSR⟨31:1⟩. (See section 13.2.3 for more details.)
2. Ensure that no more vector memory reads or writes can occur. Reading the VMAC IPR using the MFPR instruction does the required scalar/vector memory synchronization without any exceptions being reported. Reading the VMAC also ensures that all unreported hardware errors encountered by previous vector memory instructions are reported before the MFPR completes. (For more information on this function of the VMAC, refer to section 13.5.)
3. Set a software scalar-context-switch flag and perform a normal scalar processor context switch (e.g., SVPCTX) leaving the vector processor state as is.

Although not required by the architecture, software may wait for VPSR⟨BSY⟩ to be clear after disabling the vector processor when performing a scalar context switch. This has a few advantages:

1. The vector processor cannot be executing non-memory-access instructions from the previous process while a normal scalar context switch to a new process is being performed, which may be desirable to an operating system.
2. All unreported hardware errors encountered by previous non-memory-access instructions will be reported by the time the vector processor clears VPSR⟨BSY⟩ and thus known to software before scalar-context switching continues (refer to section 13.5).
3. The MFPR from the VPSR used to read VPSR⟨BSY⟩ also ensures that the scalar processor views the vector processor as disabled.

If software does not wait for VPSR⟨BSY⟩ to be clear, it is possible that while a normal scalar context switch to a new process is being performed, the vector processor may still be executing non-memory-access instructions from the previous process.

The required steps for vector context switching are discussed in section 13.2.4.

13.1.3 Overlapped Instruction Execution

To improve performance, the vector processor may overlap the execution of multiple instructions, that is, execute them concurrently. Further, when no data dependencies are present, the vector processor may complete instructions out of order relative to the order in which they were issued. A vector processor implementation can perform overlapped instruction execution by having separate function units for such operations as addition, multiplication, and memory access. Both data-dependent and data-independent instructions can be overlapped; the former by a technique known as chaining, which is described in the next section. In many instances, overlapping allows an operation from one instruction to be performed in any order with respect to an operation of another instruction.

When vector arithmetic exceptions occur during overlapped instruction execution, exception handling software may not see the same instruction state and exception information that would be returned from strictly sequential execution. Most notably, the VAER could indicate the exception conditions and destination registers of a number of vector instructions that were executing concurrently and encountered exceptions. (Exception reporting during chained execution is discussed further in the next section.)

To ensure correct program results and exception reporting, the architecture places requirements on the ordering among the operations of one vector instruction and those of another. The primary goal of these requirements is to ensure that the results obtained from both the overlapped and the strictly sequential execution of data-dependent instructions are identical. A secondary goal is to establish places within the instruction stream where software is guaranteed to receive the reporting of exceptions from a chain of data-dependent instructions.

In many cases, these requirements ensure the obvious: for example, an output vector register element of one arithmetic instruction must be computed before it can be used as an input element to a subsequent instruction. But a number of the results ensured are not so obvious: for example, an MSYNC must report exceptions encountered in generating a value of the VMR that is used in a previously issued masked store instruction.

To precisely define the requirements on the ordering among operations, section 13.1.3 discusses the "dependence" among their results (the vector register elements and control register bits produced by the operations).

Vector Chaining

The architecture allows vector chaining, in which the results of one vector instruction are forwarded (chained) to another before the input vector of the first instruction has been completely processed. In this way, the execution of data-dependent vector instructions may be overlapped. Thus, chaining is an implementation-dependent feature that is used to improve performance.

With some restrictions stated below, the vector processor may chain a number of instructions. Usually, each instruction is performed by a separate function unit. The number and types of instructions allowed within a chained sequence (often referred to as a "chain") are implementation dependent. Typically, implementations will attempt to chain sequences of two or three instructions, such as operate-operate, operate-store, load-operate, operate-operate-store, and load-operate-store. The sequence load-operate-operate-store is also possible.

The following is an example of a sequence that an implementation will often chain:

```
VVADDF  V0, V1, V2
VVMULF  V2, V3, V4
```

The destination of the VVADDF is a source of the succeeding VVMULF. The VVMULF begins executing when the first sum element of the VVADDF is available.

A number of instructions within a chained sequence can encounter exceptions. For each instruction that encounters an exception, the vector processor records the exception condition type and destination register number in the vector arithmetic exception register (VAER). When the last instruction within the chain completes, the VAER will show the exception condition type and destination register number of all instructions that encountered exceptions within the chain. Furthermore, when the vector processor disabled fault is finally generated for the exceptions, the VAER may also indicate exception state for instructions issued

after the last instruction within the chain. This effect is possible due to the asynchronous exception-reporting nature of the vector processor.

Furthermore, for each instruction that encounters an exception within a chain, the default result, as defined in section 13.2.2, is forwarded as the source operand to the next instruction. This has the effect that default results and exceptions can propagate down through a chain. Note that the default result of one instruction may be overwritten by another instruction before the exception is taken.

Consider the following:

```
VVADDG V1, V2, V3   ;gets Floating Overflow
VVGEQG V3, V4       ;gets Floating Reserved Operand
VVMULG V4, V5, V3   ;overwrites V3
```

For this example, assume that an exception is taken after the completion of the VVMULG. The VAER will indicate floating overflow and floating reserved operand exception condition types and V3 as a destination register. However, no default result will be found in the appropriate element of V3 because it has been overwritten by the VVMULG.

The architecture allows a vector load to be chained into a vector operate instruction provided the operate instruction can be suspended and resumed to produce the correct result if the vector load results in a memory management exception. Consider the following example:

```
VLDL    A, #4, V0
VVADDF  V0, V1, V1
```

In synchronous memory management mode, the VVADDF cannot be chained into the VLDL until the VLDL is ensured to complete without a memory management exception. This is because the scalar processor is not even allowed to issue the VVADDF to the vector processor until the memory management checks for the VLDL have been completed. In asynchronous memory management mode, the VVADDF may be chained into the VLDL prior to the completion of memory management exception checking. This is possible because a memory management exception in asynchronous memory management mode provides sufficient state to

restart both the VLDL and the VVADDF when the memory management exception is corrected.

The architecture allows a vector operate instruction to be chained into a store instruction. If the vector operate instruction encounters an arithmetic exception, the exception condition type and destination register number are recorded in the VAER. The default result generated by that instruction (in some cases, an encoded reserved operand) may be written to memory by the store instruction before the exception is reported.

Register Conflict

When overlapping the execution of instructions, the vector processor must deal with register conflict. Register conflict occurs when one instruction is intending to write a register while previously issued instructions are reading from that register. The following is an example of vector register conflict:

```
VVADDF  V1, V2, V3
VVMULF  V4, V5, V1
```

In this example, the VVADDF and VVMULF cannot begin execution simultaneously because the elements of V1 generated by the VVMULF would overwrite the original elements of V1 required as input by the VVADDF. However, a vector processor implementation can still overlap the execution of these two instructions in a number of ways. One way would be by not starting the VVMULF until the first element of V1 has been read by the VVADDF. In this manner, as the VVADDF reads the next elements from V1 and V2, the VVMULF writes its product into the previous element of V1. This process continues until all the elements have been processed by both instructions. The VVADDF will finish execution while the VVMULF still has at least one product to store.

In the case of the vector mask register (VMR), the vector processor ensures that register conflict does not occur. This is often accomplished by making a copy of the VMR value under which a pending vector instruction is to execute and using this copy when execution begins. This allows the vector processor to begin executing an instruction that writes to the VMR before it completes prior instructions that read from the VMR.

Dependencies Among Vector Results

To achieve correct results and exception reporting during overlapped execution, the vector processor must maintain certain dependencies among the register elements and control register bits produced by various vector instructions. Because of the vector processor's asynchronous exception reporting nature and out-of-order completion of instructions, these dependencies differ from those ensured by the VAX scalar processor. In addition, these dependencies are at the level of vector register elements and vector control register bits, rather than at the level of vector registers and vector control registers.

Among other things, these dependencies determine the exception reporting nature of the MFVP instruction. The value of the vector control register (VCR, VLR, VMR⟨31:0⟩, VMR⟨63:32⟩) delivered by an MFVP depends on the value of certain vector register elements and vector control register bits. Unreported exceptions that occur in the production of these elements and control register bits are reported by the vector processor prior to the completion of the MFVP from the vector control register.

The dependencies are expressed formally for the various classes of vector instructions by the tables of pseudocode in this section. These are the only dependencies that software should rely on the vector processor to ensure.

A vector processor implementation is allowed to ensure more than just these dependencies, providing that the larger set of dependencies yields correct results and exception reporting.

The dependencies for the various vector instructions are listed in Figures 13.1 through 13.8. For these figures, implicit in statements of the form "result DEPENDS on B" is the requirement that the result depends only on the value of "B" generated by the most immediate previously-issued instruction relative to the result's own generating instruction. For example, in

```
VVSUBF V5, V6, V7
VVADDF V1, V2, V7
VVMULF V7, V7, V3
VVDIVF V1, V4, V7
```

the V3 vector produced by the VVMULF has the dependence

```
VVADDx, VSADDx, VVSUBx, VSSUBx, VVMULx, VSMULx, VVDIVx,
VSDIVx, VVCVTxy, VVBICL, VSBICL, VVBISL, VSBISL, VVXORL,
VSXORL, VVSLLL, VSSLLL, VVSRLL, VSSRLL:

FOR i = 0 to VLR-1
   BEGIN
   Vc[i] DEPENDS on VLR
   IF {MOE EQL 1} then Vc[i] DEPENDS on VMR<i>
   IF ({MOE EQL 1} AND {VMR<i> EQL MTF}) OR {MOE EQL 0} then
      BEGIN
      Vc[i] DEPENDS on Vb[i]
      IF {Vector-Vector Operation} AND NOT {VVCVTxy} then
         Vc[i] DEPENDS on Va[i]
      END
   END
```

Figure 13.1
Dependencies for Vector Operate Instructions

```
VLDx, VGATHx:

FOR i = 0 to VLR-1
   BEGIN
   Vc[i] DEPENDS on VLR
   IF {MOE EQL 1} then Vc[i] DEPENDS on VMR<i>
   IF ({MOE EQL 1} AND {VMR<i> EQL MTF}) OR {MOE EQL 0} then
      BEGIN
      IF VGATH then
         BEGIN
         Vc[i] DEPENDS on Vb[i]
         k = BASE + Vb[i]
         END
      ELSE
         k = BASE + i * STRIDE
      Vc[i] DEPENDS on LOAD_COMPLETED(k)
      END
   END
```

Figure 13.2
Dependencies for Vector Load and Gather Instructions

```
VSTx, VSCATx:

j = 0
FOR i = 0 to VLR-1
   BEGIN
   IF ({MOE EQL 1} AND {VMR<i> EQL MTF}) OR {MOE EQL 0} then
      BEGIN
      IF {MOE EQL 1} then
         ELEMENT_STORED[j] depends on VMR<i>
      ELEMENT_STORED[j] DEPENDS on Vc[i]
      ELEMENT_STORED[j] DEPENDS on VLR
      IF VSCAT then
         BEGIN
         ELEMENT_STORED[j] DEPENDS on Vb[i]
         k = BASE + Vb[i]
         END
      ELSE
         k = BASE + i * STRIDE
      STORE_COMPLETED(k) DEPENDS on ELEMENT_STORED[j]
      j = j+1
      END
   END
```

Figure 13.3
Dependencies for Vector Store and Scatter Instructions

```
VVCMPx, VSCMPx:

FOR i = 0 to VLR-1
   BEGIN
   VMR<i> DEPENDS on VLR
   IF {MOE EQL 1} then VMR<i> DEPENDS on VMR<i>
   IF ({MOE EQL 1} AND {VMR<i> EQL MTF}) OR {MOE EQL 0} then
      BEGIN
      VMR<i> DEPENDS on Vb[i]
      IF VVCMP then VMR<i> DEPENDS on Va[i]
      END
   END
```

Figure 13.4
Dependencies for Vector Compare Instructions

```
VVMERGE, VSMERGE:

FOR i = 0 to VLR-1
   BEGIN
   Vc[i] DEPENDS on VLR
   Vc[i] DEPENDS on VMR<i>
   IF {VMR<i> EQL MTF} then
      BEGIN
      IF VVMERGE then Vc[i] DEPENDS on Va[i]
      END
   ELSE
      Vc[i] DEPENDS on Vb[i]
   END
```

Figure 13.5
Dependencies for Vector MERGE Instructions

```
IOTA:

j = 0
FOR i = 0 to VLR-1
   BEGIN
   Vc[j] DEPENDS on VLR
   IF {VMR<i> EQL MTF} then
      BEGIN
      Vc[j] DEPENDS on VMR<0..i>
      j = j+1
      END
   END
VCR DEPENDS on VMR<0..VLR-1>
```

Figure 13.6
Dependencies for IOTA Instruction

```
MSYNC:    DEPENDS on:
          - All STORE_COMPLETED(x) of previously-issued VST
            and VSCAT
          - All LOAD_COMPLETED(x)  of previously-issued VLD
            and VGATH

SYNC:     DEPENDS on the vector register elements and vector
          control register bits produced and stored by all
          previous vector instructions

MFVMRLO: DEPENDS on VMR<0..31>

MFVMRHI: DEPENDS on VMR<32..63>

MFVCR:    DEPENDS on VCR
MFVLR:    DEPENDS on VLR
```

Figure 13.7
Dependencies for MFVP Instructions

```
VSYNC:    DEPENDS on nothing, but for each memory location, x,
          forces all subsequent LOAD_COMPLETED(x) and
          STORE_COMPLETED(x) to DEPEND on all previous
          LOAD_COMPLETED(x) and STORE_COMPLETED(x)

MTVP:     DEPENDS on nothing

Value of a memory location:
          The value of a memory location DEPENDS on nothing
          and is not DEPENDED on by any vector instruction

Transitive dependence:
          IF {a DEPENDS on b} AND {b DEPENDS on c} then
             a DEPENDS on c
```

Figure 13.8
Miscellaneous Dependencies

"V3[i] DEPENDS on V7[i]." This means that the value of V3[i] produced by the VVMULF depends only on the value of V7[i] produced by the VVADDF.

13.2 Vector Processor Exceptions

There are two major classes of vector processor exceptions:

1. Vector memory management
   - Access control violation
     - Vector access control violation
     - Vector alignment
     - Vector I/O space reference
   - Translation not valid
   - Modify
2. Vector arithmetic
   - Floating underflow
   - Floating divide by zero
   - Floating reserved operand
   - Floating overflow
   - Integer overflow

   Floating underflow and integer overflow can be disabled on a per-instruction basis by clearing cntrl⟨EXC⟩.

Vector processor arithmetic exceptions cause the vector processor to disable itself (see section 13.2.3). The vector processor does not disable itself for vector processor memory management exceptions.

13.2.1 Vector Memory Management Exception Handling

Vector processor memory management exceptions are taken through the SCB vector for their scalar counterparts. Figure 13.9 illustrates the memory management fault stack frame that contains the memory management fault parameter.

- The length (L) bit, the PTE reference (P) bit, and the modify or write intent (M) bit are defined in Chapter 4. Vector processor memory management exceptions set these bits in the same way as required for scalar memory management exceptions.
- The vector alignment exception (VAL) bit must be set when an access control violation has occurred due to a vector element not being properly aligned in memory.

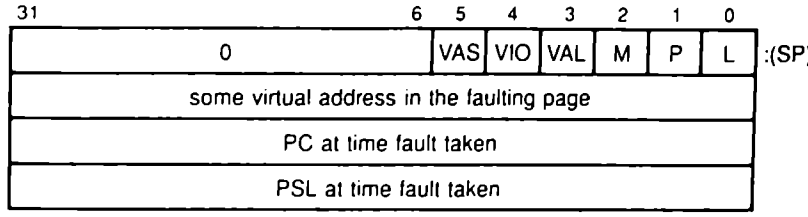


Figure 13.9
Memory Management Fault Stack Frame

- The vector I/O space reference (VIO) bit is set by some implementations to indicate that an access control violation has occurred due to a vector instruction reference to I/O space.
- The vector asynchronous memory management exception (VAS) bit must be set to indicate that a vector processor memory management exception has occurred when the asynchronous memory management scheme, described below, is implemented.

If more than one kind of memory management exception could occur on a reference to a single page, then access control violation takes precedence over both translation not valid and modify. If more than one kind of access control violation could occur, the precedence of vector access control violation, vector alignment exception, and vector I/O space reference is UNPREDICTABLE.

The architecture allows an implementation to choose one of two methods for dealing with vector processor memory management exceptions. The two methods are as follows:

- Synchronous memory management handling: restart from the beginning.
- Asynchronous memory management handling: store and reload implementation-specific state using the VSAR.

With the synchronous method, no new instructions are processed by the vector or the scalar processor until the vector memory access instruction is guaranteed to complete without incurring memory management exceptions. In such an implementation, the vector memory access instruction is backed up when a memory management exception occurs and a normal VAX memory management (access control violation, translation not valid, modify) fault taken with the PC pointing to the faulting vector memory

access instruction. If the synchronous method is implemented, the VSAR is omitted. After fixing the vector processor memory management exception, software may return to the faulting vector instruction through an REI. Alternately, software may context switch to another process. For further details on this, see section 13.2.4.

With the asynchronous method, vector memory management exceptions set VPSR⟨PMF⟩ and VPSR⟨MF⟩. The vector processor does not inform the scalar processor of the exception condition; the scalar processor continues processing instructions. All pending vector instructions that have started execution are allowed to complete if their source data is valid. The scalar processor is notified of exception condition(s) when it sends the next vector instruction to the vector processor, and a normal VAX memory management fault is taken. The saved PC points to this instruction, which is not the vector memory access instruction that incurred the memory management exception. At this point, the vector processor clears VPSR⟨PMF⟩. After fixing the vector processor memory management exception, software may allow the current scalar/vector process to continue. Before vector processor instruction execution resumes using the state that already exists in the vector processor, the vector processor clears VPSR⟨MF⟩, and the faulting memory reference is retried. Alternately, software may context switch to another process. For further details see section 13.2.4.

When a vector processor memory management exception is encountered by a VLD or VGATH instruction, the contents of the destination vector register elements are UNPREDICTABLE. When a vector processor memory management exception is encountered by a VSTL or VSCAT instruction, it is UNPREDICTABLE whether the vector processor writes any result location for which an exception did not occur. In either case, if the fault condition can be eliminated by software and the instruction restarted, then the vector processor will ensure that all destination register elements or result locations are written.

13.2.2 *Vector Arithmetic Exceptions*

Vector arithmetic exceptions are not reported like their scalar counterparts; there is no vector arithmetic exception vector in the SCB. When a vector arithmetic exception occurs, it is reported

by a vector processor disabled fault, which occurs sometime after the arithmetic exception was recognized. The PC reported by the disabled fault points to some vector instruction issued after the vector instruction that encountered the arithmetic exception; this PC value need not point to the next vector instruction issued after the one that got the exception — in practice, it seldom does. Thus, vector arithmetic exception reporting is not "precise" — the PC value reported when the disabled fault occurs is not necessarily the PC value of the vector instruction that encountered the arithmetic exception.

Vector operate instructions are always executed to completion, even if a vector arithmetic exception occurs. For each element of the destination vector register that is to be updated, if an exception occurs in computing the element's new value, then a default result is written to the vector register element. The default result is:

- The low-order 32 bits of the true result for integer overflow.
- Zero for floating underflow if exceptions are disabled.
- An encoded reserved operand for floating divide by zero, floating overflow, reserved operand, and enabled floating underflow. (See section 12.8.) For vector convert instructions that convert floating-point data to integer data, where the source element is a reserved operand, the value written to the destination element is UNPREDICTABLE.

The exception condition type and destination register number are always recorded in the VAER when a vector arithmetic exception occurs. Refer to section 12.2.3 for more information.

13.2.3 *Vector Processor Disabled*

As a result of error conditions or software control, the vector processor signals the scalar processor not to issue any more vector instructions. The vector processor is disabled when this signal is generated, and its state is reflected in VPSR⟨VEN⟩. Because the scalar and vector processors can execute asynchronously, the scalar processor may not receive this signal immediately. As a result, the scalar processor may continue to view the vector processor as enabled and send it vector instructions. Once the scalar processor receives this signal, it will view the vector processor as disabled and not send it any more vector instructions (including MFVP/MTVP). While the vector processor is disabled, and in the

absence of hardware errors, it will complete all pending instructions in its instruction queue, including those sent by the scalar processor after the vector processor became disabled.

The vector processor can either disable itself or be disabled by software. The following error conditions cause the vector processor to disable itself:

- Vector arithmetic exception (flagged by VPSR⟨AEX⟩)
- Hardware error (flagged by VPSR⟨IMP⟩ in some implementations)
- On some implementations, receipt of an illegal vector opcode (flagged by VPSR⟨IVO⟩)

In these cases, the vector processor clears VPSR⟨VEN⟩ and flags the error condition by setting the appropriate bit in the VPSR. (See Table 12.1.)

Operating system software disables the vector processor by writing a 0 into VPSR⟨VEN⟩ using an MTPR instruction. Once the vector processor is disabled, only software can enable it. The software does this by writing a 1 to VPSR⟨VEN⟩ using an MTPR. After performing an MTPR to the VPSR, software must then issue an MFPR from the VPSR to ensure that the new state of the VPSR will affect the execution of subsequently issued vector instructions. The MFPR will not complete in this case until the new state of the vector processor becomes visible to the scalar processor.

When the vector processor disables itself due to a hardware error, it is implementation-dependent whether the vector processor completes any pending vector instruction. In this case, however, the vector processor ensures when it is re-enabled that all uncompleted instructions have been flushed from the instruction queue.

If the scalar processor attempts to issue a vector instruction after it views the vector processor as disabled, then a vector processor disabled fault occurs. This fault uses the vector at SCB offset 68 (hex) and pushes the values of the PC and PSL at the time the fault was taken on to the stack. The exception handling software (running on the scalar processor) can then read the vector internal processor registers with MFPR instructions to determine what

exception conditions are recorded in the vector processor and if the vector processor is still busy processing other unfinished instructions.

Once the scalar processor views the vector processor as disabled, the only operations that can be issued to the vector processor are MTPR and MFPR to and from the vector IPRs.

13.2.4 *Handling Disabled Faults and Vector Context Switching*

The following flow outlines the required steps for handling a vector processor disabled fault.

If the new process executing on the scalar processor has a vector instruction to execute, saving and restoring the state of the vector processor, termed "vector context switching," are done as part of handling a subsequent vector processor disabled fault.

If a vector processor disabled fault occurs and the current scalar process is also the current vector process, then software must do the following:

1. Obtain the vector processor status by reading the VPSR using the MFPR instruction.
2. Perform the following checks to see if any of these conditions caused the vector processor to be disabled (If any of these conditions exist, a decision to not continue this flow may occur):
   - If VPSR⟨IVO⟩ is set, then write 1 to clear VPSR⟨IVO⟩ using the MTPR instruction and report an illegal vector opcode error.
   - If VPSR⟨IMP⟩ is set, then write 1 to clear VPSR⟨IMP⟩ using the MTPR instruction and report an implementation-specific error.
   - If VPSR⟨AEX⟩ is set, then write 1 to clear VPSR⟨AEX⟩ using the MTPR instruction and enter the vector arithmetic exception handler with information in the VAER.
3. If the software scalar-context-switch flag is set, indicating that a scalar context switch has been done, then
   - Make sure the vector processor has access to correct P0LR, P0BR, P1LR, and P1BR values.

- If any vector translation buffer needs to be invalidated, then write zero into the VTBIA IPR using the MTPR instruction. Flushing the vector translation buffer is required if the process was swapped out and the mapping change has not yet been made known to the vector translation buffer.
- Clear the software scalar-context-switch flag.

4. Enable the vector processor by writing 1 to VPSR⟨VEN⟩ using the MTPR instruction. Ensure that the new state of the vector processor becomes visible to the scalar processor by reading the VPSR with the MFPR instruction.
5. REI to retry the vector instruction at the time of the vector processor disabled fault. If there is an asynchronous memory management exception pending, it is taken when that vector instruction is re-issued to the vector processor.

If a vector processor disabled fault occurs and the current scalar process is not the current vector process, then software must do the following:

1. Check if there is a current vector process; if there is, then
   a. Wait for VPSR⟨BSY⟩ to be clear using the MFPR instruction.
   b. Check the setting of VPSR⟨IMP⟩; if it is 1 then report that the vector processor was disabled by an implementation-specific error. If this condition exists, a decision not to continue this flow may occur.
   c. If VPSR⟨IVO⟩ is set, then set a software IVO flag for this process. The illegal vector opcode error is handled when this process next tries to execute in the vector processor.
   d. If VPSR⟨AEX⟩ is set, then set a software AEX flag for this process and save vector arithmetic exception state from the VAER using the MFPR instruction. Any vector arithmetic exception conditions are handled when this process next tries to execute in the vector processor.
   e. At this point, there cannot be a synchronous memory management exception pending. But if asynchronous memory management handling is implemented, there may be an asynchronous memory management exception pending. Because scalar/vector memory synchronization was required before scalar context switching, all such pending exceptions are known at this time. So, if VPSR⟨PMF⟩ is set, then
      - Set a software asynch-memory-exception-pending flag for this process.

- Store implementation-specific vector state in memory starting at the address in the VSAR by writing one to VPSR⟨STS⟩ using the MTPR instruction.

f. Reset the vector processor state to clear the VAER and the VPSR, and enable the vector processor. Writing a 1 to both VPSR⟨RST⟩ and VPSR⟨VEN⟩ using the same MTPR instruction accomplishes this. Ensure that the new state of the vector processor becomes visible to the scalar processor by reading the VPSR with the MFPR instruction.

g. Store the current vector (V0–V15) and vector control (VLR, VMR, and VCR) register values using VST and MFVP instructions.

h. Read the VMAC IPR using the MFPR instruction. This ensures scalar/vector memory synchronization and that all hardware errors encountered by previous vector memory instructions have been reported.

2. Make the current scalar process also be the current vector process.
3. Clear the software scalar-context-switch flag.
4. Make sure the vector processor has access to correct P0LR, P0BR, P1LR, and P1BR values and invalidate any vector translation buffer by writing zero to the VTBIA IPR using the MTPR instruction.
5. Load the saved vector (V0–V15) and vector control (VLR, VMR, and VCR) register values using VLD and MTVP instructions.
6. If the software IMP, IVO, or AEX flag for this process is set, then:

   - Disable the vector processor by writing 0 to VPSR⟨VEN⟩ using the MTPR instruction. Ensure that the new state of the vector processor becomes visible to the scalar processor by reading the VPSR with the MFPR instruction.
   - If set, clear the software IMP flag for this process and finish handling the implementation-specific error. A decision not to continue this flow may occur.
   - If set, clear the software IVO flag for this process and report that an illegal vector opcode error occurred. A decision not to continue this flow may occur.
   - If set, clear the software AEX flag for this process and enter the vector arithmetic exception handler with saved VAER state. A decision not to continue this flow may occur.

7. If the software async-memory-exception-pending flag for this process is set, then
   - Clear the software async-memory-exception-pending flag for this process.
   - Send the vector processor the memory address that points to implementation-specific vector state for this process by writing the VSAR using the MTPR instruction.
   - Reload the implementation-specific vector state for this process and leave the vector processor enabled by writing 1 to both VPSR⟨RLD⟩ and VPSR⟨VEN⟩ using the same MTPR instruction. From this state, the vector processor determines if VPSR⟨PMF⟩ and/or VPSR⟨MF⟩ need to be set, and does it. Ensure that the new state of the vector processor becomes visible to the scalar processor by reading the VPSR with the MFPR instruction.

8. REI to retry the vector instruction at the time of the vector processor disabled fault. If there is an asynchronous memory management exception pending, it is taken when that vector instruction is re-issued to the vector processor.

13.2.5 *Examples of MFVP Exception Reporting*

This section gives examples of MFVP exception reporting that are ensured by the vector processor. The rules used to determine the correct result for each example are found in Figures 13.1 through 13.8, the description of MSYNC in section 13.3.2, and the description of MFVP in section 12.10.

The following four examples illustrate which exceptions are ensured by the vector processor to always cause an MSYNC to fault:

```
VVMULF    V1, V1, V2
VVADDF    V3, V2, V3
MTVLR     #1
VSTL      V2, A,  #4
VVCVTFD   V2, V3
MSYNC     R0
```

The MSYNC faults if exceptions occur in the production of V2[0] by the VVMULF or in the storage of V2[0] by the VSTL. The MSYNC need not fault if exceptions occur in the production of V2[1..VLR−1] by the VVMULF, V3[0..VLR−1] by the VVADDF, or V3[0..VLR−1] by the VVCVTFD.

```
VVADDF    V1, V1, V0
VLDL      A,  #4, V0
MSYNC     R0
```

The MSYNC faults if exceptions occur in the loading of V0[0..VLR−1] from memory. The MSYNC need not fault if exceptions occur in the production of V0[0..VLR−1] by the VVADDF.

```
VVADDF    V1, V1, V2
VLDL      A,  #4, V1
MSYNC     R0
```

The MSYNC faults if exceptions occur in the loading of V1[0..VLR−1] from memory. The MSYNC need not fault if exceptions occur in the production of V2[0..VLR−1] by the VVADDF.

```
VVMULF    V1, V1, V2
VVGTRF    V2, V3
VSTL/1    V0, A,  #4
MSYNC     R0
```

The MSYNC faults if exceptions occur in the production of V2[0..VLR−1] by the VVMULF instruction, in the production of VMR⟨0..VLR−1⟩ by the VVGTRF, or in the storage by the VSTL/1 of elements of V0 for which the corresponding VMR bit is 1.

The following six examples illustrate which exceptions the vector processor will report prior to the completion of an MFVP from a vector control register:

```
VLDL      A, #4, V1
VVMULF    V1, V1, V2
MTVLR     #1
VVGTRF    V2, V3
MFVMRHI   R1
MFVMRLO   R2
```

Unreported exceptions that occur in the loading of V1[0] from memory by the VLDL, in the production of V2[0] by the VVMULF, and in the production of VMR⟨0⟩ by the VVGTRF are reported by the vector processor prior to the completion of the MFVMRLO. The vector processor need not at that time report any exceptions that occur in the loading of V1[1..63] from memory by the VLDL or in the production of V2[1..63] by the VVMULF. Note that the vector processor need not report any exceptions before completing MFVMRHI.

```
VVGTRF    V0, V1
MTVMRLO   #patt
MFVMRLO   R1
```

For any value of i in the range of 0 to 31, inclusive: the value of VMR⟨i⟩ delivered by MFVMRLO depends only on the value placed into VMR⟨i⟩ by the MTVMRLO. As a result, the vector processor need not report exceptions that occur in the production of the VMR by the VVGTRF prior to completing the MFVMRLO.

```
VVMULF/1  V1, V1, V2
MTVMRLO   #patt
MFVMRLO   R1
```

For any value of i in the range of 0 to 31, inclusive: the value of VMR⟨i⟩ delivered by MFVMRLO depends only on the value placed into VMR⟨i⟩ by the MTVMRLO. As a result, the vector processor need not report exceptions that occur in the production of V2[0..VLR−1] by the VVMULF/1 prior to completing the MFVMRLO.

```
MTVLR     #64
VVMULF    V0, V0, V2
VVGTRF    V0, V2
MTVLR     #32
IOTA      #str, V4
MFVCR     R1
```

Prior to the completion of the MFVCR, the vector processor must report any exceptions that occurred in the production of V2[0..31] by the VVMULF and VMR⟨0..31⟩ by the VVGTRF. Note that VCR produced by an IOTA depends only on VMR⟨0..VLR−1⟩. Note that no exceptions can occur in the production of V4[0..VCR−1] by IOTA.

```
MTVLR     #64
VLDL      A, #4, V2
VVGTRF    V0, V1
VSGTRF/1  #3.0, V2
MFVMRLO   R1
```

For any value of i in the range of 0 to 31, inclusive: prior to the completion of the MFVMRLO, the vector processor must report any exceptions that occur in the loading of V2[i] from memory for which V0[i] is greater than V1[i], in the production of VMR⟨0..31⟩ by the VVGTRF, and in the production of VMR⟨0..31⟩ by the VSGTRF/1.

```
VVMULF    V1, V1, V1
VSTL      V1, base, #str
MTVMRLO   base
MFVMRLO   R1
```

In this example, the value of VMR⟨31:0⟩ delivered by MFVMRLO depends only on the value placed into VMR⟨31:0⟩ by the MTVMRLO; whether this value is V1[0] or the previous value of the location is UNPREDICTABLE. As a result, the vector processor need not report exceptions that occur in the production of V1 by the VVMULF or in the storage of V1 by the VSTL.

13.3 Synchronization

For most cases, to achieve good performance the vector processor should operate concurrently with the scalar processor. However, there are cases where the operation of the vector and scalar processors must be synchronized to ensure correct program results. Rather than force the vector processor to detect and automatically provide synchronization in these cases, the architecture provides software with special instructions to accomplish the synchronization. These instructions synchronize the following:

1. Exception reporting between the vector and scalar processors
2. Memory accesses between the scalar and vector processors
3. Memory accesses between multiple load/store units of the vector processor

Software must determine when to use these synchronization instructions to ensure correct results.

The following sections describe the synchronization instructions.

13.3.1 Scalar/Vector Instruction Synchronization (SYNC)

A mechanism for scalar/vector instruction synchronization between the scalar and vector processors is provided by SYNC, which is implemented by the MFVP instruction. SYNC allows software to ensure that the exceptions of previously issued vector instructions are reported before the scalar processor proceeds with the next instruction. SYNC detects both arithmetic exceptions and asynchronous memory management exceptions and reports these exceptions by taking the appropriate VAX instruction fault. Once it issues the SYNC, the scalar processor executes no further instructions until the SYNC completes or faults.

In beginning the execution of SYNC, the vector processor determines if any previously-issued vector instruction has encountered exceptions that have yet to be reported to the scalar processor. If so, the SYNC is faulted; otherwise, the vector processor waits for either of the following conditions to be true:

- A pending or currently executing vector instruction encounters an exception, in which case the SYNC faults; or
- The vector processor determines that all currently-executing and pending vector instructions (including memory instructions in asynchronous memory management mode) will execute to completion without encountering vector exceptions. In that case, the SYNC completes.

When SYNC completes, a longword value (which is UNPREDICTABLE) is returned to the scalar processor. The scalar processor writes the longword value to the scalar destination of the MFVP and then proceeds to execute the next instruction. If the scalar destination is in memory, it is UNPREDICTABLE whether the new value of the destination becomes visible to the vector processor until scalar/vector memory synchronization is performed.

When SYNC faults, it is not completed by the vector processor and the scalar processor does not write a longword value to the scalar destination of the MFVP. Also, depending on the exception condition encountered, the SYNC itself takes either a vector processor disabled fault or a memory management fault. If both faults are encountered while the vector processor is performing SYNC, then the SYNC itself takes a vector processor disabled fault. It is UNPREDICTABLE whether the vector processor is idle when the fault is generated. After the appropriate fault has been serviced, the SYNC can be returned to through an REI.

SYNC affects only the scalar/vector processor pair that executed it. It has no effect on other processors in a multiprocessor system.

13.3.2 *Scalar/Vector Memory Synchronization*

Scalar/vector memory synchronization allows software to ensure that the memory activity of the scalar/vector processor pair has ceased and the resultant memory writes have been made visible to each processor in the pair before the pair's scalar processor proceeds with the next instruction. Two ways ensure scalar/vector memory synchronization: using MSYNC, which is implemented

by the MFVP instruction, and using the MFPR instruction to read the vector memory activity check IPR (VMAC). The next two sections discuss MSYNC and the VMAC in detail.

Scalar/vector memory synchronization does not mean that previously issued vector memory instructions have completed; it means only that the vector and scalar processors are no longer performing memory operations. While both the VMAC and MSYNC provide scalar/vector memory synchronization, MSYNC performs significantly more than just that function. In addition, the VMAC and MSYNC differ in their exception behavior.

Note that scalar/vector memory synchronization affects only the scalar/vector processor pair that executed it. It has no effect on other processors in a multiprocessor system. Scalar/vector memory synchronization does not ensure that the writes performed by one scalar/vector pair are visible to any other scalar or vector processor.

Software can make data visible and shared between a scalar/vector pair and other scalar and vector processors by using the mechanisms described in section 7.1. Software must first make a memory write by the vector processor visible to its associated scalar processor through scalar/vector memory synchronization before making the write visible to other processors. Without performing this scalar/vector memory synchronization, it is UNPREDICTABLE whether the vector memory write will be made visible to other processors even by the mechanisms described in section 7.1.

Waiting for VPSR⟨BSY⟩ to be clear does not guarantee that a vector write is visible to the scalar processor.

Memory Instruction Synchronization (MSYNC)

MSYNC does more than perform scalar/vector memory synchronization. MSYNC allows software to ensure that all previously issued memory instructions of the scalar/vector processor pair are complete and their results made visible before the scalar processor proceeds with the next instruction.

MSYNC is implemented through the non-privileged MFVP instruction. Arithmetic and asynchronous memory management

exceptions encountered by previous vector instructions can cause an MSYNC to fault.

Once it issues an MSYNC, the scalar processor executes no further instructions until the MSYNC completes or faults. The MSYNC completes when the following three conditions have been met:

1. all previously issued scalar and vector memory instructions have completed;
2. all resultant memory writes (scalar writes and vector stores) have been made visible to both the scalar and vector processor; and
3. no exception that should cause the MSYNC to fault (as described below) has occurred.

The MSYNC faults when

1. Any unreported exception has occurred in the production or storage of any result (vector register element or vector control register bit) on which the MSYNC depends. Such results include all elements loaded or stored by a previously issued vector memory instruction as well as any element or control register bit on which these elements depend.

It is UNPREDICTABLE whether an MSYNC faults due to exceptions that occur in the production and storage of results (vector register elements and vector control register bits) on which the MSYNC does not depend. Software should not rely on such exceptions being reported by the MSYNC for program correctness.

When the MSYNC completes, a longword value (which is UNPREDICTABLE) is returned to the scalar processor, which writes it to the scalar destination of the MFVP. The scalar processor then proceeds to execute the next instruction. If the scalar destination is in memory, it is UNPREDICTABLE whether the new value of the destination becomes visible to the vector processor until another scalar/vector memory synchronization instruction is performed.

When an MSYNC faults, it is not ensured that all previously issued scalar and vector memory instructions have finished. In this case, the scalar processor writes no longword value to the scalar

destination of the MFVP. Depending on the exception encountered by the vector processor, the MSYNC takes a vector processor disabled fault or memory management fault. Note that it is UNPREDICTABLE whether the vector processor is idle when the fault is generated. After the fault has been serviced, the MSYNC may be returned to through an REI.

Section 13.1.3 gives the necessary rules and examples to determine the vector control register elements and vector control register bits on which an MSYNC depends.

Memory Activity Completion Synchronization (VMAC)
Privileged software needs a way to ensure scalar/vector memory synchronization that will not result in any exceptions being reported. Reading the VMAC IPR with the privileged MFPR instruction is provided for these situations. It is especially useful for context switching.

Once a MFPR from the VMAC is issued by the scalar processor, the scalar processor executes no further instructions until the VMAC completes, which it does when the following three conditions have been met:

1. all vector and scalar memory activities have ceased;
2. all resultant memory writes have been made visible to both the scalar and vector processor; and
3. a longword value (which is UNPREDICTABLE) is returned to the scalar processor.

After writing the longword value to the scalar destination of the MFPR, the scalar processor then proceeds to execute the next instruction. If the scalar destination is in memory, it is UNPREDICTABLE whether the new value of the destination becomes visible to the vector processor until another scalar/vector memory synchronization operation is performed.

As stated in section 13.3.2, the ceasing of vector and scalar memory activities does not mean that previously issued vector memory instructions have completed. For example, consider a vector memory instruction that has suspended execution due to an asynchronous memory management exception or hardware error. Once it becomes suspended, the instruction will write no further

elements, and its memory activity will cease. As a result, a subsequently issued MFPR from the VMAC will complete as soon as those writes that were made by the memory instruction before it was suspended are visible to both the scalar and the vector processors. But after the completion of the MFPR from the VMAC, the memory instruction is not completed and remains suspended.

Vector arithmetic and memory management exceptions of previous vector instructions never fault an MFPR-from-the-VMAC and never suspend its execution.

13.3.3 *Other Synchronization Between the Scalar and Vector Processors*

In the absence of pending vector arithmetic exceptions, reading a vector control register using the MFVP instruction waits for all previous writes to that register to complete. In addition, the scalar processor must wait for the MFVP result to be written before processing other instructions. An MFVP instruction that reads a vector control register must fault if there is any unreported exception that has occurred in the production of the value of the control register.

Writing to the VTBIA or the VSAR with MTPR causes the new state of the changed vector IPR to affect the execution of all subsequently issued vector instructions.

Reading from the VPSR with MFPR after writing to the VPSR with MTPR causes the new state of the VPSR (and the VAER if cleared by VPSR⟨RST⟩) to affect the execution of subsequently issued vector instructions.

13.3.4 *Memory Synchronization Within the Vector Processor (VSYNC)*

The vector processor may concurrently execute a number of vector memory instructions through the use of multiple load/store paths to memory. When it is necessary to synchronize the accesses of multiple vector memory instructions, the MSYNC instruction can be used; however, there are cases for which this instruction does more than is needed. If it is known that only synchronization between the memory accesses of vector instructions is required, the VSYNC instruction is more efficient.

VSYNC orders the conflicting memory accesses of vector memory instructions issued after VSYNC with those of vector memory instructions issued before VSYNC. Specifically, VSYNC forces the access of a memory location by any subsequent vector memory instruction to wait for (depend on) the completion of all prior conflicting accesses of that location by previous vector memory instructions.

VSYNC does not have any synchronizing effect between scalar and vector memory access instructions. VSYNC also has no synchronizing effect between vector load instructions because multiple load accesses cannot conflict. It also does not ensure that previous vector memory management exceptions are reported to the scalar processor.

13.3.5 *Required Use of Memory Synchronization Instructions*

Table 13.1 shows, for all possible pairs of vector or scalar reads and writes to a common memory location, whether one of the scalar/vector memory synchronization instructions or the VSYNC instruction must be issued after the first reference and before the second. Since the MSYNC instruction also includes the VSYNC function, it can always be used instead of VSYNC.

In general, these rules apply to any sequence of instructions that access a common memory location, no matter how many other vector or scalar instructions are issued between the first instruction that accesses a location and the second instruction that accesses the same location. For example, the code sequence shown below depicts a vector load followed by a scalar write to the same memory location. Between these two instructions are other scalar/vector instructions that do not access the common memory location. A scalar/vector memory synchronization instruction (MSYNC or VMAC) must be executed some time after the vector read and before the scalar write to the common location (here an MSYNC is shown).

```
VLDL   A, #4, V0
.
other instructions that do not access A
.
MSYNC  Dst
MOVL   R0, A
```

Table 13.1
When MSYNC (M), VMAC (M), or VSYNC (V) Is Needed Between Instructions That Reference the Same Memory Location

First Reference: Second Reference: Operation Sequence	Scalar Scalar	Scalar Vector	Vector Scalar	Vector Vector
Read/Read	No[a,b]	No[a]	No[a]	No[a]
Read/Write	No[b]	No[c]	M	V[e]
Write/Read	No[b]	M[d]	M	V
Write/Write	No[b]	M[d]	M	V

[a]Scalar/vector memory synchronization or VSYNC is never required between two read accesses to a memory location.
[b]Scalar/vector memory synchronization is never required between two accesses by the VAX scalar processor to a memory location.
[c]The scalar read is synchronous and will have completed before a vector memory operation is issued.
[d]Although a scalar write is a synchronous instruction, scalar/vector memory synchronization is required to ensure that the written data is visible to the vector processor before the vector memory reference is executed.
[e]See section 13.3.5 for the conditions when VSYNC is not required between a vector memory read/write pair.

In most cases, an MSYNC is the preferred method for ensuring scalar/vector memory synchronization. However, there are special cases, usually encountered by an operating system, when the VMAC is more appropriate.

Cases that require scalar/vector memory synchronization are as follows:

- After a vector instruction that stores to memory and before a peripheral (I/O) data transfer of the stored location is initiated by an application program. This ensures that the value stored will be transferred to the output device. The application must ensure that this requirement is met by using an MSYNC. Using an MFPR from the VMAC in this case is not sufficient because, unlike an MSYNC, the VMAC does not ensure that all previous vector memory instructions have successfully completed.
- After a vector instruction that stores to memory and before the associated scalar processor can execute a HALT. This ensures that a read or modify by another processor will access

the updated memory value. An MFPR from the VMAC is the preferred method for this case.

- Before the vector processor state is saved as a result of power failure. A read or modify of the same memory must read the updated value (provided that the duration of the power failure does not exceed the maximum non-volatile period of the main memory). Also, software is responsible for saving any pending vector processor exception status. An MFPR from the VMAC is the preferred method for this case.
- Before a context switch. Software is responsible for ensuring that the vector processor has completed all its memory accesses before performing a context switch. Software is also responsible for saving any pending vector processor exception status. An MFPR from the VMAC is the preferred method for this case.

The scalar/vector memory synchronization instructions are the only ones that guarantee that the memory operations of the vector and scalar processors are synchronized. Writes to I/O space, changes in access mode, machine checks, interprocessor interrupts, and execution of a HALT, REI, or interlocked instruction do not make the results of vector instructions that write to memory visible to the scalar processor, I/O subsystem, or other processors. Execution of a scalar/vector memory synchronization instruction must precede any of these mechanisms to ensure synchronization of all system components.

When VSYNC Is Not Required

Conditions exist when VSYNC is not required between conflicting vector memory accesses. A VSYNC is not required before a vector memory store instruction (VST/VSCAT) if, for each memory location to be accessed by the store, both of the following conditions are met:

1. Each of the store's accesses to the location does not conflict with any access to the location by previously issued vector store instructions. Conflict is avoided in this case because either
   - the location is not shared; or
   - all accesses to the location by previous store instructions were forced to complete by the issue of an MSYNC or an MFPR from the VMAC.

2. Each of the store's accesses to the location does not conflict with any access to the location by previously issued vector load (VLD/VGATH) instructions. Conflict is avoided in this case because of one of the following:
   - the location is not shared;
   - all accesses to the location by previous load instructions were forced to complete by the issue of an MSYNC or an MFPR from the VMAC; or
   - each of the store's accesses to the location depends on the completion (as seen by the vector processor) of all accesses to the location by previous LOAD instructions. (The examples immediately following demonstrate this concept.)

In all other cases of conflicting vector memory accesses, VSYNC is necessary to ensure correct results.

In the following examples, VSYNC is not required because both of the preceding conditions have been met for each location accessed by the store instruction:

```
VLDL        A, #4, V0          VLDL        A, #4, V0
VSTL        V0, A, #4          VSSUBL      R0, V0, V1
                               VSTL        V1, A, #4

VLDL/0      A, #4 ,V0          VLDL        A, #4, V0
VSMULL/0    #3, V0, V0         VSGTRF      #0, V0
VLDL/1      A, #4 ,V1          VLDL/1      B, #4, V1
VVMULL/1    V1, V1, V1         VLDL/0      C, #4, V2
VVMERGE/1   V1, V0, V2         VVMERGE/0   V2, V1, V3
VSTL        V2, A, #4          VSTL        V3, A, #4
```

In the following examples, VSYNC is required before the vector memory store instruction:

```
VLDL/1      A,#4,V0
VSLSSL      #0,V1
VSYNC
VSTL/1      V1,A,#4
```

If the VSYNC is not included, V0 could contain incorrect data at the end of the sequence, since the vector processor is allowed to begin the VSTL before the VLDL is finished. This is because there is no dependence between the VMR value used by the VLDL and the VSTL.

```
VLDL        A, #4, V0
VVMERGE/0   V0, V1, V1
VSYNC
VSTL        V1, A, #4
```

Unless the programmer can ensure that the VMR mask being used by the VVMERGE will force the access of each location by the VSTL to depend on the access to that location by the VLDL, a VSYNC is required. In general, when masked operates provide a conditional path of dependence between conflicting memory accesses, a VSYNC is usually necessary to ensure correct results.

```
VSTL        V1, A, #4
MTVLR       #32
VSYNC
VLDL        A+128, #4, V2
```

In this example, the VSTL writes locations A to A+255 and the VLDL reads locations A+128 to A+255. Without the VSYNC, the vector processor is allowed to start reading locations A+128 to A+255 for the VLDL before the vector processor completes (or even starts) writing locations A+128 to A+255 for the VSTL. Consequently, V2[0:31] will not contain V1[32:63], which is the intended result.

Note that the rules on when VSYNC is not required apply only to waiving the use of VSYNC prior to VST/VSCAT instructions.

```
VGATHL      A, V2, V0     ; let at least two elements
                          ; of V2 be equal
VVMULL      V9, V0, V1
VSYNC
VSCATL      V1, A, V2
```

The VSYNC is needed in this example because the VSCATL may store elements of V1 into a common location before the VGATHL has finished loading that location into all the appropriate elements of V0. As a result, elements of V0 fetched from the same location may be unequal. Suppose in the example that V2[0] = V2[63] = 0 and that the original value of location A before the sequence starts is X. Then it is possible without the VSYNC that V0[63] = X*V9[0] and that (A)= V1[63] = V9[63]*V9[0]*X after the sequence completes.

```
VLDL        A, #0, V0
VVMULL      V9, V0, V1
VSYNC
VSTL        V1, A, #0
```

The VSYNC is needed in this example because the VSTL may store elements of V1 into A before the VLDL has finished loading all elements of V0 from A. As a result, the elements of V0 may be unequal and so produce incorrect results.

13.4 *Memory Management*

The vector processor may include its own translation buffer and maintain its own copies of SBR, SLR, P0BR, P0LR, P1BR, and P1LR as a group, or may use the scalar processor's memory management unit. Hardware implementations must ensure that MTPR instructions to these registers update the copy retained by the vector processor. Changes to P0BR, P0LR, P1BR, and P1LR due to a LDPCTX do not update the copies in the vector processor. Before software enables the vector processor again, explicit MTPRs to P0BR, P0LR, P1BR, and P1LR are required to guarantee correct operation.

An MTPR to TBIS must also invalidate the corresponding TB entry in the vector processor, and an MTPR to TBIA must also invalidate the entire TB in the vector processor. However, the vector TB is not invalidated by a LDPCTX instruction. Software can use an MTPR to the vector TB invalidate all (VTBIA) register to invalidate only the vector TB. An MTPR to the VTBIA results in no operation on a processor that uses a common TB for the scalar and vector processors.

Updates to memory management registers and invalidates of TB entries in the vector processor take place even when the vector processor is disabled (VPSR<VEN> is clear). However, the vector processor may load translation buffer entries only when it is executing a vector memory access instruction.

Vector memory access instructions must not be used to read or write page tables. If a vector instruction is used to read or write page tables, the results are UNPREDICTABLE.

Vector instructions are not allowed to reference I/O space. If a vector instruction references I/O space, the results are UNPREDICTABLE.

Issuing vector instructions with memory management disabled causes the operation of the vector processor to be UNDEFINED. Disabling memory management when the vector processor is busy (VPSR⟨BSY⟩ is set) also causes the operation of the vector processor to be UNDEFINED.

13.5 *Hardware Errors*

A vector processor implementation may experience error conditions (such as chip malfunctions, parity errors, or bus errors) that prevent it from executing and completing instructions and from which it cannot recover through its own means. Such errors are termed hardware errors and may occur at any time, even when the vector processor is already disabled. Vector processor hardware errors normally do not halt the scalar processor.

At some point after the error condition occurs, the vector processor reports the error to the scalar processor. The reporting may be accomplished: through a machine check; or by disabling the vector processor, setting VPSR⟨IMP⟩, and generating a vector processor disabled fault when the next vector instruction is issued. After the error is reported, the appropriate software handler will be invoked to diagnose the vector processor and determine the severity of the hardware error and whether the vector processor can be restarted.

During execution, software may wish to force the reporting of hardware errors encountered by previous vector instructions before issuing further ones. This can be accomplished by reading the VMAC IPR and by waiting for VPSR⟨BSY⟩ to become clear.

An MFPR from the VMAC ensures that all pending vector memory instructions have finished or are suspended by an asynchronous memory management exception and that all vector processor hardware errors encountered by these instructions are reported by the time the MFPR completes.

- If the errors are reported by machine check, then the exception is taken either on the actual MFPR from VMAC or on the instruction immediately following the MFPR.
- If the errors are reported through VPSR⟨IMP⟩, the vector processor sets VPSR⟨IMP⟩ and disables itself by the time the scalar processor completes the MFPR from the VMAC. Subsequently, a vector processor disabled fault will occur when the next vector instruction is issued. A read of the VPSR immediately after the MFPR from the VMAC completes will find the vector processor disabled and VPSR⟨IMP⟩ set.

Waiting for VPSR⟨BSY⟩ to become clear before issuing further instructions ensures that all previous non-memory access instructions have been finished or are suspended by an asynchronous memory management exception and that all vector processor hardware errors encountered by these instructions are reported by the time VPSR⟨BSY⟩ becomes clear.

- If the errors are reported by machine check, then the exception is taken either on the first instruction during which the new state of VPSR⟨BSY⟩ becomes visible to the scalar processor or on the instruction immediately thereafter.
- If the errors are reported through VPSR⟨IMP⟩, the vector processor sets VPSR⟨IMP⟩ and disables itself by the time it clears VPSR⟨BSY⟩. Subsequently, a vector processor disabled fault will occur when the next vector instruction is issued. The first MFPR instruction that reads VPSR⟨BSY⟩ as clear will also read VPSR⟨VEN⟩ as clear and VPSR⟨IMP⟩ as set.

Reading the VMAC does not ensure that hardware errors encountered by pending non-memory access instructions will be reported. Waiting for VPSR⟨BSY⟩ to become clear does not ensure that vector-processor hardware errors encountered by vector memory instructions are reported.

Software can force the reporting of hardware errors encountered during the execution of previous vector instructions (both memory and non-memory) by waiting for VPSR⟨BSY⟩ to become clear and then issuing an MFPR from the VMAC. This technique can be used during scalar context switching to cause hardware errors resulting from the execution of vector instructions for the current process to be reported before that process is context switched.

A

Opcode Assignments

A.1 One-Byte Opcodes

Hex	Mnemonic	Hex	Mnemonic
00	HALT	20	ADDP4
01	NOP	21	ADDP6
02	REI	22	SUBP4
03	BPT	23	SUBP6
04	RET	24	CVTPT
05	RSB	25	MULP
06	LDPCTX	26	CVTTP
07	SVPCTX	27	DIVP
08	CVTPS	28	MOVC3
09	CVTSP	29	CMPC3
0A	INDEX	2A	SCANC
0B	CRC	2B	SPANC
0C	PROBER	2C	MOVC5
0D	PROBEW	2D	CMPC5
0E	INSQUE	2E	MOVTC
0F	REMQUE	2F	MOVTUC
10	BSBB	30	BSBW
11	BRB	31	BRW
12	BNEQ, BNEQU	32	CVTWL
13	BEQL, BEQLU	33	CVTWB
14	BGTR	34	MOVP
15	BLEQ	35	CMPP3
16	JSB	36	CVTPL
17	JMP	37	CMPP4
18	BGEQ	38	EDITPC
19	BLSS	39	MATCHC
1A	BGTRU	3A	LOCC
1B	BLEQU	3B	SKPC
1C	BVC	3C	MOVZWL
1D	BVS	3D	ACBW
1E	BGEQU, BCC	3E	MOVAW
1F	BLSSU, BCS	3F	PUSHAW

Hex	Mnemonic	Hex	Mnemonic
40	ADDF2	60	ADDD2
41	ADDF3	61	ADDD3
42	SUBF2	62	SUBD2
43	SUBF3	63	SUBD3
44	MULF2	64	MULD2
45	MULF3	65	MULD3
46	DIVF2	66	DIVD2
47	DIVF3	67	DIVD3
48	CVTFB	68	CVTDB
49	CVTFW	69	CVTDW
4A	CVTFL	6A	CVTDL
4B	CVTRFL	6B	CVTRDL
4C	CVTBF	6C	CVTBD
4D	CVTWF	6D	CVTWD
4E	CVTLF	6E	CVTLD
4F	ACBF	6F	ACBD
50	MOVF	70	MOVD
51	CMPF	71	CMPD
52	MNEGF	72	MNEGD
53	TSTF	73	TSTD
54	EMODF	74	EMODD
55	POLYF	75	POLYD
56	CVTFD	76	CVTDF
57	Reserved to DIGITAL	77	Reserved to DIGITAL
58	ADAWI	78	ASHL
59	Reserved to DIGITAL	79	ASHQ
5A	Reserved to DIGITAL	7A	EMUL
5B	Reserved to DIGITAL	7B	EDIV
5C	INSQHI	7C	CLRQ, CLRD, CLRG
5D	INSQTI	7D	MOVQ
5E	REMQHI	7E	MOVAQ, MOVAD, MOVAG
5F	REMQTI	7F	PUSHAQ, PUSHAD, PUSHAG

Hex	Mnemonic	Hex	Mnemonic
80	ADDB2	A0	ADDW2
81	ADDB3	A1	ADDW3
82	SUBB2	A2	SUBW2
83	SUBB3	A3	SUBW3
84	MULB2	A4	MULW2
85	MULB3	A5	MULW3
86	DIVB2	A6	DIVW2
87	DIVB3	A7	DIVW3
88	BISB2	A8	BISW2
89	BISB3	A9	BISW3
8A	BICB2	AA	BICW2
8B	BICB3	AB	BICW3
8C	XORB2	AC	XORW2
8D	XORB3	AD	XORW3
8E	MNEGB	AE	MNEGW
8F	CASEB	AF	CASEW
90	MOVB	B0	MOVW
91	CMPB	B1	CMPW
92	MCOMB	B2	MCOMW
93	BITB	B3	BITW
94	CLRB	B4	CLRW
95	TSTB	B5	TSTW
96	INCB	B6	INCW
97	DECB	B7	DECW
98	CVTBL	B8	BISPSW
99	CVTBW	B9	BICPSW
9A	MOVZBL	BA	POPR
9B	MOVZBW	BB	PUSHR
9C	ROTL	BC	CHMK
9D	ACBB	BD	CHME
9E	MOVAB	BE	CHMS
9F	PUSHAB	BF	CHMU

Hex	Mnemonic	Hex	Mnemonic
C0	ADDL2	E0	BBS
C1	ADDL3	E1	BBC
C2	SUBL2	E2	BBSS
C3	SUBL3	E3	BBCS
C4	MULL2	E4	BBSC
C5	MULL3	E5	BBCC
C6	DIVL2	E6	BBSSI
C7	DIVL3	E7	BBCCI
C8	BISL2	E8	BLBS
C9	BISL3	E9	BLBC
CA	BICL2	EA	FFS
CB	BICL3	EB	FFC
CC	XORL2	EC	CMPV
CD	XORL3	ED	CMPZV
CE	MNEGL	EE	EXTV
CF	CASEL	EF	EXTZV
D0	MOVL	F0	INSV
D1	CMPL	F1	ACBL
D2	MCOML	F2	AOBLSS
D3	BITL	F3	AOBLEQ
D4	CLRL, CLRF	F4	SOBGEQ
D5	TSTL	F5	SOBGTR
D6	INCL	F6	CVTLB
D7	DECL	F7	CVTLW
D8	ADWC	F8	ASHP
D9	SBWC	F9	CVTLP
DA	MTPR	FA	CALLG
DB	MFPR	FB	CALLS
DC	MOVPSL	FC	XFC
DD	PUSHL	FD	Two-Byte Opcode
DE	MOVAL, MOVAF	FE	Two-Byte Opcode
DF	PUSHAL, PUSHAF	FF	Two-Byte Opcode

A.2 Two-Byte Opcodes

Hex	Mnemonic	Hex	Mnemonic
00FD	Reserved to DIGITAL	50FD	MOVG
01FD	Reserved to DIGITAL	51FD	CMPG
02FD	WAIT	52FD	MNEGG
		53FD	TSTG
		54FD	EMODG
03FD		55FD	POLYG
to		56FD	CVTGH
2FFD	Reserved to DIGITAL	57FD	Reserved to DIGITAL
30FD	Reserved to DIGITAL	58FD	Reserved to DIGITAL
31FD	MFVP	59FD	Reserved to DIGITAL
32FD	CVTDH	5AFD	Reserved to DIGITAL
33FD	CVTGF	5BFD	Reserved to DIGITAL
34FD	VLDL	5CFD	Reserved to DIGITAL
35FD	VGATHL	5DFD	Reserved to DIGITAL
36FD	VLDQ	5EFD	Reserved to DIGITAL
37FD	VGATHQ	5FFD	Reserved to DIGITAL
38FD	Reserved to DIGITAL	60FD	ADDH2
39FD	Reserved to DIGITAL	61FD	ADDH3
3AFD	Reserved to DIGITAL	62FD	SUBH2
3BFD	Reserved to DIGITAL	63FD	SUBH3
3CFD	Reserved to DIGITAL	64FD	MULH2
3DFD	Reserved to DIGITAL	65FD	MULH3
3EFD	Reserved to DIGITAL	66FD	DIVH2
3FFD	Reserved to DIGITAL	67FD	DIVH3
40FD	ADDG2	68FD	CVTHB
41FD	ADDG3	69FD	CVTHW
42FD	SUBG2	6AFD	CVTHL
43FD	SUBG3	6BFD	CVTRHL
44FD	MULG2	6CFD	CVTBH
45FD	MULG3	6DFD	CVTWH
46FD	DIVG2	6EFD	CVTLH
47FD	DIVG3	6FFD	ACBH
48FD	CVTGB	70FD	MOVH
49FD	CVTGW	71FD	CMPH
4AFD	CVTGL	72FD	MNEGH
4BFD	CVTRGL	73FD	TSTH
4CFD	CVTBG	74FD	EMODH
4DFD	CVTWG	75FD	POLYH
4EFD	CVTLG	76FD	CVTHG
4FFD	ACBG	77FD	Reserved to DIGITAL

Hex	Mnemonic	Hex	Mnemonic
78FD	Reserved to DIGITAL	A0FD	VVMULL
79FD	Reserved to DIGITAL	A1FD	VSMULL
7AFD	Reserved to DIGITAL	A2FD	VVMULG
7BFD	Reserved to DIGITAL	A3FD	VSMULG
7CFD	CLRH, CLRO	A4FD	VVMULF
7DFD	MOVO	A5FD	VSMULF
7EFD	MOVAH, MOVAO	A6FD	VVMULD
7FFD	PUSHAH, PUSHAO	A7FD	VSMULD
80FD	VVADDL	A8FD	VSYNC
81FD	VSADDL	A9FD	MTVP
82FD	VVADDG	AAFD	VVDIVG
83FD	VSADDG	ABFD	VSDIVG
84FD	VVADDF	ACFD	VVDIVF
85FD	VSADDF	ADFD	VSDIVF
86FD	VVADDD	AEFD	VVDIVD
87FD	VSADDD	AFFD	VSDIVD
88FD	VVSUBL	B0FD	Reserved to DIGITAL
89FD	VSSUBL	B1FD	Reserved to DIGITAL
8AFD	VVSUBG	B2FD	Reserved to DIGITAL
8BFD	VSSUBG	B3FD	Reserved to DIGITAL
8CFD	VVSUBF	B4FD	Reserved to DIGITAL
8DFD	VSSUBF	B5FD	Reserved to DIGITAL
8EFD	VVSUBD	B6FD	Reserved to DIGITAL
8FFD	VSSUBD	B7FD	Reserved to DIGITAL
90FD	Reserved to DIGITAL	B8FD	Reserved to DIGITAL
91FD	Reserved to DIGITAL	B9FD	Reserved to DIGITAL
92FD	Reserved to DIGITAL	BAFD	Reserved to DIGITAL
93FD	Reserved to DIGITAL	BBFD	Reserved to DIGITAL
94FD	Reserved to DIGITAL	BCFD	Reserved to DIGITAL
95FD	Reserved to DIGITAL	BDFD	Reserved to DIGITAL
96FD	Reserved to DIGITAL	BEFD	Reserved to DIGITAL
97FD	Reserved to DIGITAL	BFFD	Reserved to DIGITAL
98FD	CVTFH	C0FD	VVCMPL
99FD	CVTFG	C1FD	VSCMPL
9AFD	Reserved to DIGITAL	C2FD	VVCMPG
9BFD	Reserved to DIGITAL	C3FD	VSCMPG
9CFD	VSTL	C4FD	VVCMPF
9DFD	VSCATL	C5FD	VSCMPF
9EFD	VSTQ	C6FD	VVCMPD
9FFD	VSCATQ	C7FD	VSCMPD

Hex	Mnemonic
C8FD	VVBISL
C9FD	VSBISL
CAFD	Illegal Vector Opcode
CBFD	Illegal Vector Opcode
CCFD	VVBICL
CDFD	VSBICL
CEFD	Illegal Vector Opcode
CFFD	Illegal Vector Opcode
D0FD	Reserved to DIGITAL
D1FD	Reserved to DIGITAL
D2FD	Reserved to DIGITAL
D3FD	Reserved to DIGITAL
D4FD	Reserved to DIGITAL
D5FD	Reserved to DIGITAL
D6FD	Reserved to DIGITAL
D7FD	Reserved to DIGITAL
D8FD	Reserved to DIGITAL
D9FD	Reserved to DIGITAL
DAFD	Reserved to DIGITAL
DBFD	Reserved to DIGITAL
DCFD	Reserved to DIGITAL
DDFD	Reserved to DIGITAL
DEFD	Reserved to DIGITAL
DFFD	Reserved to DIGITAL
E0FD	VVSRLL
E1FD	VSSRLL
E2FD	Illegal Vector Opcode
E3FD	Illegal Vector Opcode
E4FD	VVSLLL
E5FD	VSSLLL
E6FD	Illegal Vector Opcode
E7FD	Illegal Vector Opcode
E8FD	VVXORL
E9FD	VSXORL
EAFD	Illegal Vector Opcode
EBFD	Illegal Vector Opcode
ECFD	VVCVT
EDFD	IOTA
EEFD	VVMERGE
EFFD	VSMERGE

Hex	Mnemonic
F0FD	Reserved to DIGITAL
F1FD	Reserved to DIGITAL
F2FD	Reserved to DIGITAL
F3FD	Reserved to DIGITAL
F4FD	Reserved to DIGITAL
F5FD	Reserved to DIGITAL
F6FD	CVTHF
F7FD	CVTHD
F8FD	Reserved to DIGITAL
F9FD	Reserved to DIGITAL
FAFD	Reserved to DIGITAL
FBFD	Reserved to DIGITAL
FCFD	Reserved to DIGITAL
FDFD	Reserved to DIGITAL
FEFD	Reserved to DIGITAL
FFFD	Reserved to DIGITAL
00FE to FFFE	Reserved to DIGITAL
00FF to F7FF	Reserved to DIGITAL
F8FF	Reserved to DIGITAL
F9FF	Reserved to DIGITAL
FAFF	Reserved to DIGITAL
FBFF	Reserved to DIGITAL
FCFF	Reserved to DIGITAL
FDFF	BUGL
FEFF	BUGW
FFFF	Reserved for all time

A.3 Vector Opcodes

Hex	Mnemonic	Hex	Mnemonic
31FD	MFVP	ABFD	VSDIVG
34FD	VLDL	ACFD	VVDIVF
35FD	VGATHL	ADFD	VSDIVF
36FD	VLDQ	AEFD	VVDIVD
37FD	VGATHQ	AFFD	VSDIVD
80FD	VVADDL	C0FD	VVCMPL
81FD	VSADDL	C1FD	VSCMPL
82FD	VVADDG	C2FD	VVCMPG
83FD	VSADDG	C3FD	VSCMPG
84FD	VVADDF	C4FD	VVCMPF
85FD	VSADDF	C5FD	VSCMPF
86FD	VVADDD	C6FD	VVCMPD
87FD	VSADDD	C7FD	VSCMPD
88FD	VVSUBL	C8FD	VVBISL
89FD	VSSUBL	C9FD	VSBISL
8AFD	VVSUBG	CAFD	Illegal Vector Opcode
8BFD	VSSUBG	CBFD	Illegal Vector Opcode
8CFD	VVSUBF	CCFD	VVBICL
8DFD	VSSUBF	CDFD	VSBICL
8EFD	VVSUBD	CEFD	Illegal Vector Opcode
8FFD	VSSUBD	CFFD	Illegal Vector Opcode
9CFD	VSTL	E0FD	VVSRLL
9DFD	VSCATL	E1FD	VSSRLL
9EFD	VSTQ	E2FD	Illegal Vector Opcode
9FFD	VSCATQ	E3FD	Illegal Vector Opcode
		E4FD	VVSLLL
A0FD	VVMULL	E5FD	VSSLLL
A1FD	VSMULL	E6FD	Illegal Vector Opcode
A2FD	VVMULG	E7FD	Illegal Vector Opcode
A3FD	VSMULG	E8FD	VVXORL
A4FD	VVMULF	E9FD	VSXORL
A5FD	VSMULF	EAFD	Illegal Vector Opcode
A6FD	VVMULD	EBFD	Illegal Vector Opcode
A7FD	VSMULD	ECFD	VVCVT
A8FD	VSYNC	EDFD	IOTA
A9FD	MTVP	EEFD	VVMERGE
AAFD	VVDIVG	EFFD	VSMERGE

B

Implementation Dependencies

The VAX family of processors shares a common architecture, including data types, instructions, addressing modes, and registers. Software written to depend only on these features will run on any VAX processor. Some software, however (typically operating system software), by necessity depends on features that vary from implementation to implementation.

This appendix briefly describes features that may differ from one VAX processor to another and are typically of interest to operating systems programmers. Such features include:

- Instruction set
- Layout of physical memory
- System control block
- Codes for the halt conditions
- Internal processor registers
- Contents of the machine-check stack frame
- Operations that are specified as either UNDEFINED or UNPREDICTABLE.

Due to the large number of VAX systems available, this appendix can provide only a brief description of each VAX system. For more complete and current information, refer to the appropriate *VAX Systems Hardware Handbook*, available through any Digital sales representative.

B.1 Implementation Options

Some instructions, data types, and processor registers described in this book may be omitted from VAX processors. For more detail, refer to Chapter 11.

B.1.1 The Physical Address Space

VAX virtual addresses are 32 bits in length. Virtual addresses are translated into physical addresses by a function that is implementation dependent, and that depends on whether memory mapping is enabled, as described in Chapter 4.

VAX physical addresses are at most 34 bits in length, but processors may implement fewer address bits. The physical address space consists of two parts: memory space and I/O space. Memory space starts at address zero and continues to an implementation-dependent limit. I/O space begins at that limit and continues to the end of the physical address space. Neither memory space nor I/O space is necessarily filled; typically each will be sparsely filled.

Both memory space and I/O space are addressed by bytes. Aligned and unaligned references to memory of byte, word, and longword size are supported. Only aligned longword references are necessarily supported to I/O space. References of other sizes may be supported on some implementations.

Typically, I/O space consists of several "adapter spaces" and one or more address spaces. The adapter spaces are sections of the address space set aside for the registers of various bus adapters and memory controllers. Many adapter spaces begin with an "adapter configuration register," which contains an adapter type code. This is for use by the operating system during power-up initialization to help it determine the system hardware configuration.

On systems with a UNIBUS interconnect, UNIBUS address spaces are sections of the I/O address space that directly map to a UNIBUS address space. UNIBUS addresses are 18 bits in length, so a UNIBUS address space is 256 kilobytes in length. Within the UNIBUS address space, the low 248 kilobytes is UNIBUS memory space. Typically, UNIBUS references to UNIBUS memory space are translated by a set of UNIBUS map registers to references in

the VAX physical address space. This allows UNIBUS devices to directly access VAX physical memory.

B.1.2 The System Control Block

The system control block is a block of physical memory that contains vectors for exceptions and interrupts. Chapter 5 describes its format and interpretation. VAX processors may include exception and interrupt vectors in addition to those described in Chapter 5.

B.1.3 Halt Codes

Chapter 10 describes halting. When a VAX processor halts, the reason for the halt is saved in a halt code. A processor may report halt codes in addition to those described in Chapter 10.

B.1.4 Internal Processor Registers

Chapter 8 describes the internal processor register (IPR) address space and the registers found there on every machine. Processors may include IPRs in addition to those described in Chapter 8.

B.1.5 Machine Checks

Chapter 5 describes the overall format of the machine-check stack frame. Space for implementation-dependent error report information is included in the stack frame. The circumstances that cause machine-check are different for each processor, and the information reported is different as well.

B.1.6 UNPREDICTABLE

As used in this book, the term "UNPREDICTABLE" has a particular meaning. Results specified as UNPREDICTABLE may vary from one execution to the next. Software must not depend on any UNPREDICTABLE results. The results of an instruction include

- Explicit destination operands (those with operand specifiers)
- Implicit destination operands
- Registers modified by operand specifier evaluation, including specifiers for implied operands
- PSL condition codes
- PSL⟨FPD⟩
- PSL⟨TP⟩, if PSL⟨T⟩ was set at the beginning of the instruction
- PTE⟨M⟩ for pages mapping write or modify type operands (PTE⟨M⟩ will be set if the instruction modified the page or if PTE⟨M⟩ was set before the instruction started.)

PC and unlisted fields of the PSL are specifically excluded from this list. They are UNPREDICTABLE only when they appear as explicit or implicit operands.

UNPREDICTABLE results are constrained by memory mapping and access protection. That is, if correctly operating instructions cannot affect a memory location or privileged register, then an instruction with UNPREDICTABLE results cannot either.

B.1.7 *UNDEFINED*

As used in this book, the term "UNDEFINED" has a particular meaning. UNDEFINED operations result from privileged software performing proscribed actions. The effects may be widespread and are not necessarily constrained by memory mapping or access control. UNDEFINED operations may affect the contents of memory, the operation of peripherals, and the operation of the processor. UNDEFINED operations are constrained only to not hang the processor and console. Control of the processor can be regained by reinitializing the processor from the console.

The complete list of UNDEFINED operations is implementation dependent but includes the following operations:

- Console START or CONTINUE after an error halt and before a processor initialization.
- LDPCTX when the new kernel stack is invalid or inaccessible.
- When the operation of the VAX scalar processor becomes UNDEFINED, so does the operation of its associated vector processor. The converse is not true; when the operation of the vector processor becomes UNDEFINED, the operation of the scalar processor need not become UNDEFINED.
- If the console microprocessor cannot complete its own power-up initialization, the state of the console and that of the processor is UNDEFINED.
- If the SCBB points to I/O space or non-existent memory when an exception or interrupt occurs, the operation of the processor is UNDEFINED. If SCBB contains a virtual address, the page or pages containing the SCB must be valid and accessible to kernel mode; otherwise the operation of the processor is UNDEFINED.
- If part or all of the system page table resides in I/O space, in non-existent memory, or beyond the end of the physical-

address space while memory mapping is enabled, the operation of the processor is UNDEFINED.

- If part or all of either process page table is mapped into I/O space or non-existent memory while memory mapping is enabled, the operation of the processor is UNDEFINED.
- References to PCB entries located in I/O space result in UNDEFINED behavior.
- References in the I/O space other than in UNIBUS spaces are UNDEFINED with respect to interlocking. This includes the BBSSI and BBCCI instructions.
- Any string, quadword, octaword, F_floating, D_floating, G_floating, H_floating, and field references in the I/O space result in UNDEFINED behavior.
- References to addresses beyond the end of the physical address space result in UNDEFINED behavior.
- Probing non-existent memory (or addresses beyond the end of the physical address space) results in UNDEFINED processor operation.
- If the instruction stream comes within 512 bytes of non-existent memory or the end of the physical address space when memory management is disabled, prefetching references by the processor may cause UNDEFINED behavior.
- Before enabling memory management, software must flush the entire translation buffer. If software does not follow this rule, the operation of the processor becomes UNDEFINED when memory management is enabled.
- Issuing vector instructions with memory management disabled causes the operation of the vector processor to be UNDEFINED. Disabling memory management when the vector processor is busy (VPSR⟨BSY⟩ is set) also causes the operation of the vector processor to be UNDEFINED.
- SLR, P0LR, and P1LR values not in the range of 0 to 200000 (hex), inclusive, are reserved values and result in UNDEFINED operation.
- The operation of the processor is UNDEFINED if process space page tables are read-only or no-access.
- In kernel mode, the operation of the processor is UNDEFINED after the execution of an MTPR to P0BR or P1BR that writes a non-zero value to P0BR⟨8:0⟩.
- If the processor holds a virtual address in PCBB, then the PCB must not cross a page boundary, the page containing the PCB

must be valid, the page must allow kernel write access, and (on processors that implement modify fault) the page must be marked "modified;" otherwise the operation of the processor is UNDEFINED.

- After software changes a valid system PTE that maps any part of a process page table, software must flush the translation for the system page and then flush the translations of all valid process pages so mapped. If software writes such a process page before flushing its translation, the operation of the processor is UNDEFINED.
- If IPL is lowered to zero when the processor is running on the interrupt stack, the operation of the processor is UNDEFINED.
- For kernel-stack-not-valid abort and machine-check exception, if the SCB exception vector ⟨1:0⟩ is not 1, the operation of the processor is UNDEFINED.
- Bits ⟨1:0⟩ in the CHMx and emulation exception vectors must be zero, or the operation of the processor is UNDEFINED.
- If writable control store does not exist or is not loaded, when an exception is reported through an SCB vector with bits ⟨1:0⟩ equal to 2, then the operation of the processor is UNDEFINED.
- The operation of the processor becomes UNDEFINED when servicing an exception or interrupt whose SCB vector has bits ⟨1:0⟩ both set.
- In kernel mode, the operation of the processor is UNDEFINED after execution of MTPR to a read-only register, MTPR to a non-existent register, MTPR of a non-zero value to an MBZ field, or MTPR of a reserved value to a register.
- In kernel mode, the operation of the processor is UNDEFINED after execution of MFPR from a register that does not exist or after execution of MFPR from a write-only register.
- The TBIS, TBIA, and TBCHK processor registers are write only. The operation of MFPR from any of these registers is UNDEFINED.
- Execution of *MTPR #src,#PR$_ASTLVL* with src⟨31:0⟩ GEQU 5 results in UNDEFINED behavior.
- If VPSR⟨RST⟩ is set by software while VPSR⟨BSY⟩ is set, the operation of the vector processor is UNDEFINED.

B.2 Implementation

The following sections describe all VAX systems existing as of this writing.

B.2.1 VAX-11/780

The VAX-11/780 computer system, announced in 1978, was the first processor of the VAX family. It is packaged in a cabinet 60 inches tall and 47 inches wide. The VAX-11/780 includes all the instructions (G_floating and H_floating instructions are available as an option), all the architecturally defined processor registers, and compatibility mode.

The VAX-11/780 is implemented with off the shelf Schottky TTL components on more than 20 circuit boards. It uses over 6K words of 99-bit wide microcode running at a 200 ns cycle time. It uses a 8 KB cache. 32-bit F_floating and 64-bit D_floating operations can be sped up with an optional Floating Point Accelerator. It uses the popular Unibus and Massbus, introduced on the PDP-11 family, for connection to I/O devices.

B.2.2 VAX-11/750

The VAX-11/750 computer system, announced in 1980, was the second processor in the VAX family. It is packaged in a cabinet 42 inches tall and 29 inches wide. The VAX-11/750 includes all the instructions (G_floating and H_floating are available as an option), all architecturally defined processor registers, and compatibility mode.

The VAX-11/750 is implemented with custom TTL gate array technology. It uses over 6K words of 80-bit wide microcode running at a 320 ns cycle time. It uses a 4 KB cache. 32-bit F_floating and 64-bit D_floating operations can be sped up with an optional Floating Point Accelerator. It also uses the popular Unibus and Massbus for connection to I/O devices.

It achieves approximately 60% the performance of the VAX-11/780, i.e. 0.6 VUPs.

B.2.3 VAX-11/730

The VAX-11/730 computer system, announced in 1982, was the third processor in the VAX family, and the first to include

G_floating and H_floating as standard. It is packaged with two disks in a cabinet 42 inches tall and 22 inches wide. The VAX-11/730 includes all the instructions, all the architecturally defined processor registers, and compatibility mode.

The VAX-11/730 is implemented with off the shelf TTL bit sliced arithmetic elements and programmed array logic components. It uses 16K words of 24-bit wide microcode running at a 290 ns cycle time. It does not use a cache. All floating point operations can be sped up with an optional Floating Point Accelerator. It also uses the Unibus for connection to I/O devices.

It achieves about 25% of the performance of a VAX-11/780.

B.2.4 *VAX-11/725*

The VAX-11/725 computer system, announced in 1984, is a repackaged version of the VAX-11/730 processor. The cabinet is 25 inches high and 18 inches wide, and includes memory, two tape cartridge drives, and a disk.

B.2.5 *VAX-11/782*

The VAX-11/782 computer system, announced in 1982, is a dual processor VAX-11/780 with shared memory. The cabinets containing the processor, I/O adapters, and shared memory are 60 inches tall and 190 inches wide.

B.2.6 *MicroVAX-I*

The MicroVAX-I computer system is the first subset VAX. Announced in 1984, it is packaged in a box about 6 inches by 28 inches by 22 inches. It uses a custom data path chip and off the shelf TTL components. It uses the Qbus, originally introduced with the LSI-11, for I/O.

The MicroVAX-I comes in two versions; one includes F_floating and G_floating instructions, the other includes F_floating and D_floating instructions. Neither version includes H_floating instructions. The MicroVAX-I processor includes some of the optional string instructions (CMPC3, LOCC, SCANC, SKPC, and SPANC) but does not include any of the optional processor registers or PDP-11 compatibility mode. It uses 8K words of 39-bit wide microcode running at a 250 ns cycle time.

B.2.7 VAX-11/785

The VAX-11/785 computer system, announced in 1984, was designed as a field upgrade of the VAX-11/780. Most of the logic in the VAX-11/785 is identical to the VAX-11/780, except that it uses faster TTL parts that enable it to run at a 133 ns cycle time. It uses a 32 KB cache. The memory and I/O subsystems are identical to those used in the VAX-11/780. It achieves about 1.5 times the performance of the VAX-11/780.

B.2.8 VAX 8600

The VAX 8600 computer system, announced in 1984, is packaged in a cabinet 60 inches tall and about 80 inches wide. It is the first VAX to be implemented with custom ECL gate array technology. It is a pipelined implementation with independent instruction decode, integer execution, floating point, and memory units. It features a 16 KB cache, optional floating point accelerator, and I/O subsystem taken from the VAX-11/780. It uses 8K words of 86-bit wide writable control store running at 80 ns.

The VAX 8600 includes all the instructions, architecturally defined processor registers, and compatibility mode.

It achieves about 4 times the performance of a VAX-11/780.

B.2.9 MicroVAX-II

The MicroVAX-II computer system is the first VAX with the processor on a single chip. F_floating, D_floating, and G_floating instructions are provided by a floating-point unit (another chip). The MicroVAX-II is a subset VAX, and includes none of the optional string instructions, optional processor registers, H-floating instructions, or PDP-11 compatibility mode. The CPU and FPU are implemented in custom 3 micron NMOS technology running at 200 ns cycle time. It uses 1600 words of 39-bit wide microcode.

The MicroVAX-II achieves about 90% of the performance of the VAX-11/780 on a single circuit board. It does not use a cache, and uses the Qbus for I/O.

B.2.10 VAX 8200

The VAX 8200 computer system, announced in 1986, is packaged with two disks in a cabinet 42 inches tall and 22 inches wide. It is implemented using a custom VLSI NMOS chip set that runs at

a 200 ns cycle time. It includes a 8 KB cache. It introduced the VAXBI bus as its system and I/O bus. The VAXBI operates at a 200 ns cycle.

The VAX 8200 includes all the instructions and architecturally defined processor registers, but does not include compatibility mode. It uses 16K words of 39-bit wide microcode.

It achieves about the same performance as the VAX-11/780.

B.2.11 *VAX 8300*

The VAX 8300 computer system, announced in 1986, is a dual-processor version of the VAX 8200, packaged in the same cabinet.

B.2.12 *VAX 8650*

The VAX 8650 computer system, announced in 1985, is a faster field upgrade of the 8600. The 8650 is packaged in the same cabinet as the 8600 and offers higher performance. It operates at a 55 ns cycle time and achieves about 6 times the performance of a VAX-11/780.

B.2.13 *VAX 8800*

The dual-processor VAX 8800 computer system, announced in 1986, uses the same ECL gate array technology used in the VAX 8600. It features a micro-pipelined implementation in which multiple micro-instructions are in different stages of execution simultaneously. It includes a 64 KB cache and operates at a 45 ns cycle time. It uses 16K words of 144-bit wide writable control store. It is packaged in a cabinet 60 inches tall and about 80 inches wide. It uses multiple VAXBI buses for I/O.

The VAX 8800 includes all the instructions and architecturally defined processor registers. The 8800 does not include PDP-11 compatibility mode.

Each processor in the VAX 8800 achieves about 6 times the performance of VAX-11/780, thus yielding about 12 times the throughput of a VAX-11/780 in multistream environments. The CPU used in the VAX 8800 is also used in the VAX 8700, VAX 8550, VAX 8810/8820/8830/8840. A slightly modified version is used in the VAX 8500 and VAX 8530.

B.2.14 VAX 8700

The VAX 8700 computer system, announced in 1986, is a single-processor version of the VAX 8800. It can be upgraded in the field to a VAX 8800 in the same cabinet. It achieves about 6 times the performance of a VAX-11/780.

B.2.15 VAX 8550

The VAX 8550 computer system, announced in 1986, is a single-processor version of the VAX 8800, which cannot be upgraded to a multiprocessor VAX 8800. It is packaged in a cabinet 60 inches tall and about 27 inches wide. It also achieves about 6 times the performance of a VAX-11/780.

B.2.16 VAX 8500

The VAX 8500 computer system, announced in 1986, is a slower version of the VAX 8550. It also achieves about 3 times the performance of a VAX-11/780.

B.2.17 VAX 8250

The VAX 8250 computer system, announced in 1987, is a faster version of the VAX 8200, running at a 160 ns cycle time. It achieves about 20% higher performance than the VAX 8200.

B.2.18 VAX 8350

The VAX 8350 computer system, announced in 1987, is a dual-processor version of the VAX 8250, packaged in the same cabinet.

B.2.19 VAX 8530

The VAX 8530 computer system, announced in 1987, is a faster version of the VAX 8500. It also achieves about 4 times the performance of a VAX-11/780.

B.2.20 MicroVAX 2000

The MicroVAX 2000 computer system, announced in 1987, uses the same CPU/FPU as the MicroVAX-II. It uses a tabletop package and uses SCSI for I/O. It is also available as a workstation called VAXstation 2000.

B.2.21 VAX 8810/8820/8830/8840

The VAX 88x0 computer system is a modified version of the VAX 8800 that can be expanded from 1 to 4 processors. The base CPU is identical to the VAX 8700.

B.2.22 *MicroVAX 3500/3600*

The MicroVAX 3500/3600 computer system, announced in 1987, uses the first CMOS single chip VAX microprocessor (CVAX). The CVAX chip is designed in a 2.0 micron CMOS process and runs at a 90 ns cycle time in these systems. The CPU chip includes a 1 KB cache. These systems use a 64 KB second level cache and achieve about 2.7 times the performance of a VAX-11/780. Like all other members of the MicroVAX family, they use the Qbus for I/O.

B.2.23 *MicroVAX 3300/3400*

The MicroVAX 3300/3400 computer system, announced in 1988, use the same CPU/FPU chips as the MicroVAX 3500/3600, but does not use a secondary cache. It achieves about 2.5 times the performance of a VAX-11/780. These systems are packaged in a different package and offer slightly different mass storage configurations.

B.2.24 *VAX 6200*

The VAX 6200 computer system, announced in 1988, uses the CVAX CPU/FPU chips running at 80 ns cycle time. It uses a 256 KB cache. This system introduces the XMI bus as a system bus. Up to 4 single board CPU's can be plugged in to the XMI backplane to allow the system to be configured as a multiprocessor system. The VAXBI is used for I/O.

Each CPU achieves about 2.8 times the performance of a VAX-11/780.

B.2.25 *MicroVAX 3800/3900*

The MicroVAX 3800/3900 computer system, announced in 1988, uses a faster version of the CVAX chip implemented in 1.5 micron CMOS technology operating at a 60 ns cycle time. These systems achieve about 4 times the performance of the VAX-11/780.

B.2.26 *VAX 6300*

The VAX 6300 computer system, announced in 1988, provides an upgrade to the VAX 6200 in the same cabinet. It uses a new CPU board based on a 60 ns version of CVAX in 1.5 micron CMOS technology. The packaging, memory, and I/O systems are identical to the VAX 6200. Up to 6 processors are supported.

Each processor achieves about 4 times the performance of a VAX-11/780.

B.2.27 *MicroVAX 3100*

The MicroVAX 3100 computer system, announced in 1989, uses the CVAX CPU/FPU running at 90 ns cycle time. It achieves about 2.5 times the performance of the VAX-11/780.

B.2.28 *VAX 6000 Model 400*

The VAX 6000 Model 400 computer system, announced in 1989, uses the same packaging, memory, and I/O as the VAX 6200 and VAX 6300. It provides an easy upgrade to the VAX 6200 and VAX 6300 in the same cabinet. It uses a new CPU board featuring new 1.5 micron CMOS CPU and FPU chips. It has a two level cache: 2KB on chip and 128KB off chip. Cycle time is 28ns. It uses 1.7K words of 45 bit wide microcode.

The system can be configured with up to 6 processors, each achieving about 7 times the performance of a VAX-11/780.

This VAX system is the first one to offer integrated vector processing. A single board vector coprocessor can be attached to a scalar processor. Up to 2 scalar/vector pairs are supported. The vector coprocessor runs at a 45 ns cycle and achieves peak megaflop rates of 45 megaflops in double precision, and 90 megaflops in single precision. The vector coprocessor has its own 1 MB cache.

B.2.29 *VAX 9000*

The VAX 9000 computer system, announced in 1989, is the fastest VAX computer system to date. It uses mainframe pipelined design techniques using ECL gate arrays and advanced multichip packaging technology to operate at a 16 ns cycle time. It uses a 128 KB cache. The system can be configured with up to 4 processors, each with an optional vector unit. Each scalar CPU achieves 30 to 40 times the performance of a VAX-11/780. Each vector coprocessor has a peak rating of 125 megaflops. It features the XMI bus as an I/O bus.

B.2.30 *VAX 4000-300*

The VAX 4000 computer system, announced in 1990, is based on the 1.5 micron CMOS chip set used in the VAX 6000-400. It includes built in, high performance adapters for storage and

networking, plus Qbus for other I/O. It uses a two level cache: 2KB on chip, 128KB off chip. It achieves about 8 times the performance of the VAX-11/780.

B.2.31 *VAXstation I*

With the introduction of the MicroVAX I in 1984, it became feasible, both from a cost and engineering perspective, to enhance the VAX product line with a single user, graphics-based workstation known as the VAXstation I. The VAXstation I takes the existing MicroVAX I packaging and adds to it a one bit frame buffer attached to a 1024 × 864 monochrome monitor. Input/output is accomplished through both a keyboard and mouse. The frame buffer remaps a portion of VAX physical memory to the frame buffer and software routines manipulate this memory to create a basic windowing system.

B.2.32 *VAXstation II and VAXstation II/GPX*

With the MicroVAX II, graphics capabilities were also included. The same monochrome frame buffer found on the VAXstation I is found on the MicroVAX II platform creating the VAXstation II. In January 1986 Digital introduced its first graphics processor-based subsystem, the GPX. The VAXstation II/GPX is an 1024 × 864 8-bit color or grey-scale graphics subsystem using a Digital developed graphics processor. This graphics subsystem performs the rendering portion of the graphics pipeline.

B.2.33 *VAXstation 2000*

In February 1987, Digital introduced its first desktop workstation, the VAXstation 2000. It was based on the same processor found in the MicroVAX II. In addition to its desktop form factor this product was not based on a previously introduced MicroVAX system. This was the first VAX-based system designed from the outset as a workstation. The VAXstation 2000 supported a monochrome frame buffer and a four bit color or grey-scale graphics subsystem. Later an eight bit graphics subsystem was added. This product also supported the industry standard SCSI interconnect for disks and tapes.

B.2.34 *VAXstation 3200/3500*

Introduced in July 1987, the VAXstation 3200/3500 uses the same CPU as the MicroVAX 3500/3600. The VAXstation 3200 was packaged in a small deskside pedestal while the VAXstation 3500

used the same deskside enclosure as the MicroVAX 3500. The VAXstation 3200/3500 uses the same graphics options found on the VAXstation II and VAXstation II/GPX.

B.2.35 *VAXstation 8000*

The VAXstation 8000 (VS8000), introduced in February 1988, was Digital's first true 3D workstation. Based on the VAX 8200 CPU packaged in a deskside enclosure, the VAXstation 8000 used a sophisticated 3D graphics subsystem designed with Evans and Sutherland. This graphics subsystem consisted of the complete 3D graphics pipeline and additional hardware support for such features as hardware anti-aliasing of vectors and depth queuing.

B.2.36 *VAXstation 3100*

In January 1989, Digital introduced the successor to the VAXstation 2000, based on the CVAX running at 90 ns, known as the VAXstation 3100. The VAXstation 3100 supports both a monochrome frame buffer and an 8 bit Dragon graphics subsystem. The VAXstation 3100 came in two variants. The Model 30 is a small desktop enclosure that supported two internal SCSI-based mass storage devices. The Model 40 supported five internal SCSI-based devices. In October 1989 the 60 ns CVAX CPU/FPU was added to the VAXstation 3100 family. The Models 38 and 48 represented the new CPU/FPU combination and enclosures. In January 1990 Digital further enhanced the VAXstation 3100 family with the introduction of the SPX graphics option. This graphics option was designed with the X window system in mind and offered up to 10 times the performance of GPX graphics.

B.2.37 *VAXstation 3520/3540*

This is a multiprocessor VAXstation introduced in January 1989. The VAXstation 3520 offered two CVAX CPUs, while the 3540 offered four. The clock speed was 80 ns. In addition to supporting multiprocessing the VAXstation 3520/3540 also offered follow on 3D graphics capability to the VAXstation 8000 and the first 1280 × 1024 monitor.

Index

Aborts, defined, 247
Absolute queues, 110–111
  data type, 11–12
ACB (add, compare, and branch instruction), 74–75, 130
Access-control-violation fault, 225, 226, 238, 280
  defined, 257
  stacks and, 273
Access modes, 213, 223–224
  asynchronous system traps and, 290–291
  changing, 239–240
  executive, 223
  kernel, 223
  supervisor, 224
  user, 224
  in a vector processor, 487
Access type, 26
  address, 26, 27, 41
  branch, 26, 27, 41
  instruction interpretation and, 40–41
  modify, 26, 41
  read, 26, 40
  variable-length bit field base address, 26
  write, 26, 40
Adapter configuration register, 532
Adapter spaces, 532
ADAWI (add aligned word interlocked instruction), 48, 304–306
ADC (add carry instruction), 351
ADD (add instruction). *See* Add instructions
Add instructions
  ADAWI (add aligned word interlocked), 48, 304–306
  ADC (add carry), 351
  ADD (add), integer, 49
  ADD (add), compatibility mode, 356
  ADD (add), floating-point, 130, 133
  ADDP (add packed), 178–179
  ADWC (add with carry), 50
  VADDL (vector add), integer, 454
  VADD (vector add), floating point, 463
Add, compare, and branch instruction (ACB), 74–75, 130
Add one and branch less than or equal instruction (AOBLEQ), 76
Add one and branch less than instruction (AOBLSS), 77
Address access type, 26, 27, 41. *See also* Access type
Address arguments, validating, 240–243
Addresses
  probing, 240–243
  virtual, 4
Addressing modes, 4
  autodecrement deferred mode, 339–340
  autodecrement mode, 33–34
  autoincrement deferred mode, 32–33, 339
  autoincrement mode, 29–31, 338–339
  branch mode addressing formats, 27, 39–40
  compatibility mode, 337–340
  general mode addressing formats, 27–39
  index deferred mode, 340
  index mode, 340
  notation for, 27
  register deferred mode, 337–338
  register mode, 337
  vector instruction format, restrictions on, 443–444
Address instructions
  MOVA (move address), 67
  PUSHA (push address), 68
Address translation, 219–223
Adjust input length operator (EOSADJUST_INPUT), EDITPC, 204
ADWC (add with carry instruction), 50
Alignment
  ADAWI (add aligned word interlocked), 48, 304–306
  branch destinations, 74
  interlocking and, 303, 305
  I/O registers, 319
  stack, 274
  vectors, 239, 445, 498
AOBLEQ (add one and branch less than or equal instruction), 76
AOBLSS (add one and branch less than instruction), 77
Application-extension instruction group, 419–420
Argument pointer (AP), 20
Argument validation, 239–243
Arithmetic exceptions, 254–257
  vector, 500–501

Arithmetic operators, operation description notation and, 47
Arithmetic shift instructions
ASH (arithmetic shift), 51
ASH (arithmetic shift), compatibility mode, 362
ASHC (arithmetic shift combined), 363
ASHP (Arithmetic shift and round packed), decimal string, 180–181
ASL (arithmetic shift left), 350
ASR (arithmetic shift right), 349
VSL (vector shift logical instruction), 460
Arrays, indexed mode addressing of, 39
ASH (arithmetic shift instruction). *See* Arithmetic shift instructions
ASL (arithmetic shift left instruction), 350
ASR (arithmetic shift right instruction), 349
Assembler notation, vector, 482–484
AST (asynchronous system traps), 290–292
delivery interrupt (IPL 2), 292
software interrupt, 291
ASTLVL (AST level register), 290–292
Asynchronous memory management handling, 499–500
Asynchronous system traps (AST), 290–292
Atomicity of shared data, 302–303
Atomic operands, 303
Atomic operations, 300
changing PTE, 223
PTE⟨M⟩ updates, 222
read, 300
read-modify-write, 300
write, 300
Autodecrement mode addressing, 33–34
compatibility mode, 339
deferred mode, compatibility mode, 339–340
indexed mode, 39
operand specifier format, 33–34
Autoincrement mode addressing, 29–31
compatibility mode, 338–339
deferred indexed mode, 39
deferred mode, 32–33, 339
deferred mode operand specifier format, 32–33
indexed mode, 39
operand specifier format, 29–31
Autorestart, 386

B (branch on condition instruction), 78–79
compatibility mode, 365
Base instruction group, 418
Base operand address (BOA), 38
Base operand specifier, in index mode addressing, 38–39
Base registers, 19–20, 287
BB (branch on bit instruction), 80
and modify interlocked, 82–83
and modify without interlock, 81
BBCCI (branch on bit clear and clear interlocked instruction), 82–83, 304–306
BBSSI (branch on bit set and set interlocked instruction), 82–83, 304–306
BIC (bit clear instructions). *See* Bit clear instructions
Binary load and unload console command (X), 407–408
BIS (bit set instructions). *See* Bit set instructions
BIT (bit test instruction), 53
compatibility mode, 361
Bit clear instructions
BIC (bit clear), 52
BIC (bit clear), compatibility mode, 361
BICPSW (bit clear PSW), 99
VBIC (vector bit clear), 459
Bit set instructions
BIS (bit set), 53
BIS (bit set), compatibility mode, 360
BISPSW (bit set PSW), 100
VBIS (vector bit set), 459
Bit field. *See* Variable-length bit field
Bit test instruction (BIT), 53
compatibility mode, 361
Blank backwards when zero operator (EOSBLANK), EDITPC, 205
Blauw, G. A., viii
BLB (branch on low bit instruction), 84
BOOT console command, 379–380, 392
Bootstrap-in-progress flag, 380
Bootstrapping, 379–381
BPT (breakpoint instruction), 101, 261
BPT instruction fault, compatibility mode, 375
BR (branch instructions). *See* Branch instructions
Branch displacement access type, 26, 27, 41 *See also* Access type
Branch instructions
B (branch on condition), 78–79
B (branch on condition), compatibility mode, 365
BR (branch), 84
BR (branch), compatibility mode, 364
BSB (branch to subroutine instruction), 85
Branch on bit instructions
BBC (branch on bit clear), 80
BBS (branch on bit set), 80
BBCC (branch on bit clear and clear without interlock), 81
BBCS (branch on bit clear and set without interlock), 81
BBSC (branch on bit set and clear without interlock), 81
BBSS (branch bit set and set without interlock), 81
BBCCI (branch on bit clear and clear interlocked), 82–83, 304–306
BBSSI (branch on bit set and set interlocked), 82–83, 304–306
BLB (branch on low bit), 84
Branch mode addressing, 26, 27, 39–40
Breakpoint fault, defined, 261–262
Breakpoint instruction (BPT), 101, 261
Brooks, F.P., Jr., viii
Brunner, R.A *See* Long vacation, in need of
BSB (branch to subroutine instruction), 85
BUG (bugcheck instruction), 102
Bugcheck instruction (BUG), 102
Byte
8-bit, 4
data type, 4
notation for, 26
in registers, 20
Byte branch mode addressing, 39–40
Byte displacement mode addressing, 34–35
deferred indexed mode, 39
deferred mode operand specifier format, 35
indexed mode, 39
operand specifier format, 34–35

C (carry condition code, PSL⟨C⟩), 24
Caches, 315–316

Cached PTEs, 235
Calculate cyclic redundancy check instruction (CRC), 171–174
CALL (call instructions), 91–92
  CALLG (call procedure with general argument list), 91, 93–94
  CALLS (call procedure with stack argument list), 95–96
"Can't" messages, in console mode, 411
Carriage return console command, 388–389
Carry condition code (PSL(C)), 24
CASE (case instruction), 86–87
CC (condition code operators instruction), compatibility mode, 372
Chaining, 489, 490–492
Change Mode instruction (CHMx), 283, 285–286
Change modes, 239–240
Change-mode trap, defined, 260–261
Character string
  data type, 13
  memory references and, 313–314
Character-string instructions, 153–170
  CMPC (compare characters), 154–157
  LOCC (locate character), 158
  MATCHC (match characters), 159–160
  memory references for, 313–314
  MOVC (move character), 161–163
  MOVTC (move translated characters), 164–165
  MOVTUC (move translated until character), 166–167
  SCANC (scan characters), 168
  SKPC (skip character), 169
  SPANC (span characters), 170
CHMx (change mode instruction), 283, 285–286
CLR (clear instruction), 54, 346
  floating-point, 130, 134
CM (compatibility mode bit, PSL(CM)), 22, 336–378
CMP (compare instruction). *See* Compare instructions
COM (complement instruction), 348. *See also* MCOM
Command syntax, in console I/O mode, 390–391
Comment line console command (!), 409
Communication, interprocessor, 299–306
Compare instructions
  CMP (compare), integer, 54, 70
  CMP (compare), compatibility mode, 357
  CMP (compare), floating point, 130, 135
  CMPC (compare characters), 154–157
  CMPP (compare packed), decimal string, 182
  VCMP (vector compare), integer, 455
  VCMP (vector compare), floating point, 464–465
Compatibility mode (PDP-11)
  addressing modes, 337–340
  architecture subsetting and, 421
  branch instructions, 364–365
  CM (compatibility mode bit, PSL(CM)), 22, 336–378
  compatibility-mode registers, 373
  double operand instructions, 355–363
  entering and leaving, 373
  exceptions, 260, 374–375
  faults, 375
  general registers, 337–340
  instructions, 341–366
  interrupts, 374–375
  I/O references, 377
  jump instructions, 367–368
  memory management, 373–374
  native-mode registers, 373
  odd address error abort, 374
  processor status word, 341
  process registers, 377
  program synchronization, 377–378
  registers, 373
  return from interrupts and traps instructions, 369
  single operand instructions, 345–354
  stack, 340
  subroutine instructions, 367–368
  tracing in, 375–377
  unimplemented PDP-11 traps, 377
Complement instruction (COM), 348
Compute index instruction (INDEX), 104–105
Conditional branch instruction (B). *See* Branch Instructions
Condition code operators instruction (CC), compatibility mode, 372
Condition codes, 21, 444
  operation description notation and, 47
! console command (comment line), 409
@ console command (indirect command), 409
Console I/O mode, 387–413
  commands, 391–409
  command language subsets, 410
  command syntax, 390–391
  errors and error messages, 411–412
  halts and halt messages in, 412–413
  options, 411
  special characters, 388–389
  terminal registers, 414–415
  vector support, 410
Context switching, 287
  per-process registers and, 323–327
  required use of memory synchronization instructions and, 517
  scalar, 487–488
  vector, 503–506
CONTINUE console command, 392
Control block
  in character-string instructions, 153
  in decimal-string instructions, 175
Control instructions, 74–90
  ACB (add and compare branch), 74–75
  AOBLEQ (add one and branch less than or equal), 76
  AOBLSS (add one and branch less than), 77
  B (condition branch), 78–79
  BB (branch on bit), 80
  BB (branch on bit interlocked), 82–83
  BB (branch on bit and modify without interlock), 81
  BLB (branch on low bit), 84
  BSB (branch to subroutine), 85
  CASE (case), 86–87
  JMP (jump), 88
  JSB (jump to subroutine), 88
  RSB (return from subroutine), 89
  SOBGEQ (subtract one and branch greater than or equal), 89
  SOBGTR (subtract one and branch greater than), 90
Control (P1) region, 215, 217
Control variable, interlock, 304–306
Control word operand, vector, 438–444

Convert instructions
CVT (convert), integer, 55
CVT (convert), floating-point, 136–138
CVTLP (convert long to packed), 183
CVTPL (convert packed to long), 184
CVTPS (convert packed to leading separate numeric), 185–186
CVTPT (convert packed to trailing numeric), 176, 195–196
CVTSP (convert leading separate numeric to packed), 177, 189
CVTTP (convert trailing numeric to packed), 177, 190–191
VVCVT (vector convert), 466–467
Corruption of shared data, 302–303
using interlocks to prevent, 303–306
CPUID (CPU identification register), 327
CRC (calculate cyclic redundancy check instruction), 171–174
CTRL/C console command, 390
CTRL/O console command, 389
CTRL/P console command, 390
CTRL/Q console command, 389
CTRL/S console command, 389
CTRL/U console command, 389
CUR_MOD (current access mode field, PSL⟨CUR_MOD⟩), 23
Current access mode (PSL⟨CUR_MOD⟩), 23
Current frame pointer, 19
CVT (convert instruction). *See* Convert instructions
Cyclic redundancy check instruction (CRC), 171–174

Data
incoherency, 302–303
shared. *See* Shared data
visibility, 299, 301–302
Data types, 26–27
definitions of, 4–19
in operand specifier notation, 45
DEC (decrement instruction), 56, 346
Decimal overflow, 176
Decimal overflow trap enable bit (PSL⟨DV⟩), 23, 91
Decimal string
data types, 17–19
overflow trap, 256
signs and digits of, 17–18
zero-length, 177–178
Decimal-string instructions, 175–197
ADDP (add packed), 178–179
ASHP (arithmetic shift and round packed), 180–181
CMPP (compare packed), 182
CVTLP (convert long to packed), 183
CVTPL (convert packed to long), 184
CVTPS (convert packed to leading separate numeric), 185–186
CVTPT (convert packed to trailing numeric), 176, 195–196
CVTSP (convert leading separate numeric to packed), 177, 189
CVTTP (convert trailing numeric to packed), 177, 190–191
DIVP (divide packed), 192–193
MOVP (move packed), 194
MULP (multiply packed), 195
packed decimal operations, 177
SUBP (subtract packed), 196–197
zero numbers in, 176
DECnet, x
Decrement instruction (DEC), 56, 346
Default result, vector 433, 461, 501
Dependencies, vector results, 493–497
DEPOSIT console command, 393–398
Device interrupts, 251
D_floating
data type, 7
instructions, 128–129
notation, 26
in registers, 21
vector instructions, 461
Disabled fault, vector processor, 501–506
Displacement mode addressing, 34–35
deferred mode, 35
deferred mode operand specifier formats, 35
operand specifier formats, 34–35
DIV (divide instruction). *See* Divide instructions
Divide-by-zero trap, defined, 256
Divide instructions
DIV (Divide), integer, 57
DIV (Divide), compatibility mode, 359
DIV (Divide), floating-point, 130, 131, 139
DIVP (divide packed instruction), 192–193
EDIV (extended divide), 58
VDIV (vector divide) 468–469
Double error trap, compatibility mode, 377
Double operand instructions, compatibility mode, 355–363
Double-precision floating point. *See* D_floating; G_floating
DV (decimal overflow trap enable bit, PSL⟨DV⟩), 23, 91

Edit instructions
EDITPC (edit packed to character string), 198–212
EOS (EDITPC pattern operators). *See* EOS
vector, 474–476
EDITPC (edit packed to character string instruction), 198–212
EDIV (extended divide instruction), 58
EMOD (extended multiply and integerize instruction), 130, 140–141
EMT instruction fault, compatibility mode, 375
EMUL (extended multiply instruction), 59
Emulated-only instruction group, 420
Emulation instruction exceptions, 154
Emulation software, 426
End edit operator (EOSEND), EDITPC, 205
End floating sign operator (EOSEND_FLOAT), EDITPC, 206
Entry mask, 91
EOS_ (EDITPC pattern operators)
EOSADJUST_INPUT (adjust input length), 204
EOSBLANK_ZERO (blank backwards when zero), 205
EOSEND (end edit), 205
EOSEND_FLOAT (end floating sign), 206
EOSFILL (store fill), 206
EOSFLOAT (float sign), 207
EOSINSERT (insert character), 208
EOSLOAD (load register), 209
EOSMOVE (move digits), 210
EOSREPLACE_SIGN (replace sign when zero), 211

EOS_SIGNIF (significance), 212
EOSSTORE_SIGN (store sign), 212
Error halt, processor initialization after, 384
Error notification interrupts, 248
Errors and error messages
console mode, 411–412
serious system failure, 265–266
ESP (executive-mode stack pointer), 273
EXAMINE console command, 399–400
EXC (vector control word exception enable bit, cntrl(EXC)), 439–440
Exception-enable bits, 21
vector, 439–440
Exceptions, 254–266
arithmetic, 254–257
classes of, 254
compatibility mode, 374–375
defined, 244
detected during operand reference, 258–259
disabled, 248
enabled, 248
initiating, 277–282
instructions related to, 283–286
interactions among, 267–268
versus interrupts, 247–249
memory management, 257–258
occurring as result of instruction, 259–262
system failures, 265–266
trace faults, 262–265
vectors in the SCB, 270–272
Exceptions, vector processor, 498–509
arithmetic, 500–501
memory management, 498–500
MFVP, examples, 507–509
vector, during chained execution, 489
Exception types, 246–247
Exclusive-OR instruction (XOR), 66
compatibility mode, 360
VXOR (vector exclusive-or), 459
Execute microinstruction console command (MICROSTEP), 403
Execute next VAX instruction console command (NEXT), 404
Executing on the interrupt stack, 244
Execution model, vector, 485–498
Executive access mode, 213, 223
Executive-mode stack pointer (ESP), 273
accessing, 327
Exponents, in floating point numbers, 129
Extended divide instruction (EDIV), 58
Extended function call instruction (XFC), 109
Extended multiply instruction (EMUL), 59
Extended multiply and integerize instruction (EMOD), floating-point, 140–141
Extended range double precision floating point. *See* G_floating
Extended range quadruple precision floating point. *See* H_floating
EXT (extract field instruction), 71

Faults
compatibility mode, 375
defined, 247
memory management and protection, 238–239
precedence of, 238
restartability after, 249–250
trace, 262–265
vector processor disabled, 501–506
FF (find first instruction), 73
F_floating
data type, 7, 26
instructions, 128–129
in registers, 20
vector instructions, 461
Field. *See* Variable-length bit-field
FIELD macro, 69
FIND console command, 401
Find first instruction (FF), 73
First part done bit (PSL(FPD)), 22
Floating divide-by-zero fault, defined, 256, 257
Floating point
conversion to, from integer arithmetic types, 130
data formats, 128
data types, 7–9
overflow, ACB and, 75
overflow fault, 131, 256, 257
representation of, 128–130
underflow, ACB and, 75
underflow fault, 75, 131, 257
Floating-point instructions, 128–152
accuracy of, 131–132
ADD, 133
CLR (clear), 130, 134
CMP (compare), 130, 135
CVT (convert), 136–138
DIV (divide), 139
EMOD (extended multiply and integerize), 130, 140–141
MNEG (move negated), 142
MOV (move), 130, 143
MUL (multiply), 144
overview of, 130–131
programming considerations, 132
POLY (polynomial evaluation), 145–150
SUB (subtract), 151
TST (test), 130, 152
vector, 461–473
Floating underflow enable bit (PSL(FU)), 23
floating-point underflow faults and, 131
Float sign operator (EOSFLOAT), 207
FPD (first part done bit, PSL(FPD)), 22
Frame pointer (FP), 19
FU (floating underflow enable, PSL(FU)). *See* Floating underflow enable bit

Gather memory data into vector register instruction (VGATH), 448–449
General mode addressing, 26, 27–39
General-purpose registers (GPRs), 19–21
compatibility mode, 337–340
Generate compressed iota vector instruction (IOTA), 475–476
G_floating
data type, 7–9
instructions, 128–129
notation for, 26
in registers, 21
vector instructions, 461
Global page table, 233
GPRs (general purpose registers), 19–21
Granularity, interlock 304–305

HALT (halt instruction), 103
HALT console command, 401

Halts and halt messages, in console mode, 412–413
  codes, implementation options, 533
  console command, 401
Hardware errors, vector, 521–522
Hardware initialization, after powerfail, 384–385
H_floating
  data type, 9
  instructions, 128–129
  notation, 26
  in registers, 21

ICCS (interval clock control status register), 333–335
ICR (interval count register), 333–335
Illegal addressing mode faults, 37–38
Illegal command messages, in console mode, 411–412
Illegal instruction fault, compatibility mode, 375
Illegal vector opcodes, 444
Immediate mode addressing, 29
Implementation dependencies, 531–545
  halt codes, 533
  internal processor registers, 533
  machine checks, 533
  physical address space, 532
  system control block, 533
  UNDEFINED, 534–536
  UNPREDICTABLE, 533–534
INC (increment instruction), 59, 347
Incoherency of data, 302–303
Increment instruction (INC), 59, 347
INDEX (compute index instruction), 104–105
Index mode addressing, 37–39, 340
  deferred mode addressing, 340
  operand specifier format, 37–39
Index registers, 19
  restrictions on, 39
Indirect console command (@), 409
Initialize exception or interrupt, 277–281
Initialization, after powerfail, 384–385
INITIALIZE console command, 384, 402
Initialize processor console command (INITIALIZE), 402
Initialize system bus console command (UNJAM), 406
INS (insert instructions). *See* Insert instructions
Insert instructions
  INSQHI (insert entry into queue at head, interlocked), 112–114, 304
  INSQTI (insert entry into queue at tail, interlocked), 115–117, 304
  INSQUE (insert entry in queue), 110–111, 118–119
  INSV (Insert field), 69, 73
  EOSINSERT (Insert character operator), EDITPC, 208
Instruction emulation
  architecture subsetting and, 421–424
  defined, 260
  exceptions, 426–428
Instruction execution, overlapped, 489–498
Instruction faults, 247
  compatibility mode, 375
  on a processor that sets PTE⟨M⟩, 221
Instruction format, 25
  vector, 438–444
Instruction groups
  application-extension, 419–420
  base, 418
  emulated-only, 420
Instruction interpretation, 40–42
  by hardware, 40–42
  by software, 251–253, 426
Instruction memory, 307
Instruction operands, 47
Instructions, 43–212
  compatibility mode, 341–366
  components of, 307
  interlocked, 299, 300, 304–306
  memory references made by, 311–315
  prefetching, 284, 308
  restartability of, 249–250
  usable to reference I/O space, 321–322
  vector, 445–481
Instruction-set options, subsetting, 416–421
Instruction stream
  modification, xiv, 308–311
  processing, 307–308
INSV (insert field instruction), 69, 73
Integer
  conversion to floating, 130
  data types, 4–6
  divide-by-zero trap, 255
  instructions, 48–66
  vector instructions, 454–458
Integer overflow
  ACB and, 75
  AOBLEQ and, 76
  AOBLSS and, 77
  overflow enable bit (PSL⟨IV⟩), 24, 91
  overflow trap, defined, 255
  SOBGEQ and, 89
  SOBGTR and, 90
Interlock grain, 304, 305
Interlocking
  instructions, 299, 300, 304–306
  preventing corruption of shared data with, 303–306
  queue instructions, 111, 112–117, 120–125
  rules applying to, 304–306
  sequences, 304–306
  unintentional, 305
Internal processor register (IPR). *See* IPR
Interprocessor signaling, 299, 301–302
Interrupt priority level (IPL) register. *See* IPL
Interrupts, 250–254
  compatibility mode, 374–375
  defined, 244
  device, 251
  versus exceptions, 247–249
  initiating, 277–282
  instructions related to, 283–286
  interactions among, 267–268
  process-scheduling, 292
  requesting, 299
  restartability after, 249–250
  software, 251–253
  urgent, 251
Interrupt stack bit (PSL⟨IS⟩), 23, 273–275
Interrupt-stack-not-valid halt, 265, 266
Interrupt stack pointer (ISP), 273
  accessing, 327
Interrupt vectors in the SCB, 270–272
Interval clock control status register (ICCS), 333–335
Interval clock registers, 333–335
Interval count register (ICR), 333–335

I/O, 299–322
devices, 300, 316–319
PTEs for I/O devices, 316–317
references, compatibility mode, 377
registers, 319–321
space, 299, 321–322, 532
structure, 316–322
IOTA (generate compressed iota vector instruction), 475–476
IOT instruction fault, compatibility mode, 375
IPL (interrupt priority level) register, 23, 244–246, 253–254
interrupt types and, 251–253
IPL 2 (AST delivery interrupt), 292
IPL 3 (process-scheduling interrupt), 292
IPR (internal processor register), 19, 23–24, 273, 323–335, 432–437
address space, 323
architecture subsetting and, 421–422
implementation options, 533
IS (interrupt stack bit, PSL(IS)), 23, 273–275
ISP (interrupt stack pointer), 273
IV (integer overflow enable bit, PSL(IV)), 24. *See also* Integer overflow

JMP (jump instruction), 88
compatibility mode, 367
JSB (jump to subroutine instruction), 88
JSR (jump to subroutine instruction), compatibility mode, 368

Kernel access mode, 213, 223
stack pointer (KSP), 273
Kernel stack, 273–275
Kernel-stack-not-valid abort, 251, 273–274, 280
defined, 265, 266
Kernel stack pointer (KSP), 273
accessing, 327
KSP (kernel-mode stack pointer), 273, 327

LDPCTX (load process context instruction), 293–295, 297
Leading-separate numeric strings
data type, 15–17
signs and digits of, 16–17
Length registers, 217, 224–225, 287
Length violation, 224–225
indication of, 238
Literal mode addressing, 35–37
operand specifier format, 35–37
LOAD console command (load file into memory), 402
Load file into memory console command (LOAD), 402
Load memory data into vector register instruction (VLD), 446–447
Load process context instruction (LDPCTX), 293–295, 297
Load register operator (EOSLOAD), EDITPC, 209
Load and start primary bootstrap console command (BOOT), 392
Locate character instruction (LOCC), character string, 158, 314
LOCC (locate character instruction), character string, 158, 314
Logical instructions, 48–66
shift, 459–460
vector, 459–460
Logical operators, operation description notation and, 47
Long vacation, in need of. *See* Brunner, R.A.
Longword
data type, 6
for linking queues, 11, 13, 110, 111
in registers, 20
notation for, 26
Longword displacement mode addressing, 34–35
deferred indexed mode, 39
deferred mode operand specifier format, 35
indexed mode, 39
operand specifier format, 34–35

M (page-table entry modify bit, PTE(M)), 220–222
Machine-check exception, defined, 266
Machine checks, 251
implementation options, 533
Major system state transitions, 385–387
MAPEN internal processor register, 217
Masked vector operations, 438–439
cntrl(MOE) (masked-operation enable bit), 438–439
cntrl(MTF) (match true/false bit), 438–439
Match characters instruction (MATCHC), character string, 159–160
MATCHC (match characters instruction), character string, 159–160
MAXU (maximum unsigned notation), 47
MBZ (must be zero), 3, 231, 234
MCOM (move complemented instruction), 60
Memory, 299–322
above top-of-stack, 276
access control, 213
multiprocessing and, 299–306
protection, 213–214, 223–226
Memory activity completion synchronization (VMAC), 513–514
Memory instruction synchronization (MSYNC), 478, 511–513, 514, 515, 516
Memory management, 213–243
architecture subsetting and, 422
asynchronous, 499–500
compatibility mode, 373–374
control, 217
defined, 213–214
disabled, 217–219, 521
faults, 238–239
rtVAX, 422–424
synchronous, 499–500
vector, 520–521
virtual address space and, 214
Memory management exceptions, 257–258
handling, vector processor, 498–500
Memory mapping, 213–214
disabling, 217–219
enabling, 217
Memory-mapping-enable (MME) bit, 217
Memory references, 311–315
to I/O space, 321–322
Memory space, in physical address space, 532
Memory synchronization within the vector processor (VSYNC), 481, 514–515
when not required, 517–520
MFP (move from previous space instruction), compatibility mode, 371

MFPR (move from processor register instruction), 24, 252–253, 323–327, 329
  accessing stack registers and, 275
MFVP (move from vector processor instruction), 477–479, 487, 493
  exception reporting, examples, 507–509
MI (vector control word modify-intent bit, cntrl(MI)), 440
MICROSTEP console command (execute microinstruction), 403
MINU (minimum unsigned notation), 47
MME (memory-mapping-enable) bit, 217
MNEG (move negated instruction), 60
  floating-point, 142
Mode
  access, 213, 223–224
  change, 239–240
  compatibility, 336–378
  native, 373
Modify access type, 26, 41. *See also* Access type
Modify bit, PTE(M), 220–222
Modify fault, 226, 238
  defined, 220–222, 257–258
  stacks and, 273
Modify intent bit, cntrl(MI), 440
Modify operand, 303
MOE (vector control word masked-operation enable bit, cntrl(MOE)), 438–439
MOV (move instruction). *See* Move instructions
Move instructions
  EOSMOVE (move digits operator), EDITPC, 210
  MOV (move), 61
  MOV (move), compatibility mode, 356
  MOV (move), floating-point, 130, 143
  MOVA (move address), 67
  MOVC (move character), 161–163
  MCOM (move complemented), 60
  MNEG (move negated), integer, 60
  MNEG (move negated), floating-point, 142
  MOVP (move packed), 194
  MOVTC (move translated characters), 164–165
  MOVTUC (move translated until character), 166–167, 314
  MOVZ (move zero-extended), 61
Move instructions, special purpose
  MFP (move from previous space), compatibility mode, 371
  MFPR (move from processor register), 24, 323–327, 329
  MFVP (move from vector processor), 477–479, 487
  MOVPSL (move from PSL instruction), 106
  MTP (move to previous space), compatibility mode, 370
  MTPR (move to processor register), 24, 252–254, 275, 323–328, 488
  MTVP (move to vector processor), 480
MSYNC (memory instruction synchronization), 478, 489, 511–513, 514, 515, 516
  faults, 506–507
MTF (vector control word match true/false bit, cntrl(MTF)), 438–439
MTP (move to previous space instruction), compatibility mode, 370
MTPR (move to processor register instruction), 24, 252–254, 275, 323–328, 488
MTVP (move to vector processor instruction), 480
MUL (multiply instruction). *See* Multiply instructions
Multiple events, serialization of notification of, 267–268
Multiply instructions
  EMOD (extended multiply and integerize), 130, 140–141
  EMUL (extended multiply), 59
  MUL (multiply), integer, 62
  MUL (multiply), compatibility mode, 358
  MUL (multiply), floating-point, 144
  MULP (multiply packed), 195
  VMULL (vector multiply), integer, 457
Multiprocessors
  instruction stream modification and, 310
  interrupt priority levels in, 246
  memory and, 299–306
  page table entry and, 223
Mutual exclusion protocols, 300
  implementing with interlocked instructions, 304–306

N (negative condition code, PSL(N)) 24
Native-mode registers, 373
NEG (negate instruction), 347. *See also* MNEG
Negative condition code (PSL(N)), 24
Negate instruction (NEG), 347
NEXT console command (execute next VAX instruction), 404
NICR (next interval count register), 333–334
No-access, 224
NOP (no operation instruction), 106
Notation, for addressing modes, 27
Numeric string
  leading-separate, 15–17
  trailing, data type, 13–15

OA (operand address notation), 27
Octaword
  data type, 6
  notation for, 26
  in registers, 21
Odd address error abort, compatibility mode, 374
Omitted instructions, 128, 154
  emulation of, 424, 425–426
Opcode assignments, 523–530
  one-byte opcodes, 523–526
  two-byte opcodes, 527–529
  vector opcodes, 530
Opcode formats, 25
Opcode-reserved-to-customers fault, defined, 259–260
Operand address notation (OA), 27
Operands, atomic, 303
Operand specifier format
  autodecrement mode, 33–34
  autoincrement deferred mode, 32–33
  autoincrement mode, 29–31
  displacement deferred, 35
  displacement mode, 34–35
  index mode, 37–39
  literal mode, 35–37
  register deferred mode, 29
  register mode, 28
Operand specifiers, 25–27
  compatibility mode, 337
  in general mode addressing format, 27–39

memory references for, 312–313
notation, 44–45
vector instruction format, 443–444
Operand specifier types
access types, 26
data types, 26–27
Operation description notation, 45–47
Overflow, 248
condition code (PSL⟨V⟩), 24
decimal, 176
decimal overflow enable bit (PSL⟨DV⟩), 91
decimal string overflow trap, 256
floating point, 75
floating point overflow fault, 131, 256, 257
integer, 75–77, 89, 90
integer overflow enable bit (PSL⟨IV⟩), 91
integer overflow trap, 255
stack, 265
stack overflow trap, 377
vector, 439–440, 462, 498
Overlapped instruction execution, 489–498
register conflict and, 492
Overpunched numeric encoding, 15
OWN, (owner field, PTE⟨OWN⟩, 222

P0 and P1 page tables
P0PT (P0 page table), 230–231
P1PT (P1 page table), 232–233
P0 and P1 regions
P0 (program) region, 215, 217, 227–228, 230–232
P1 (control) region, 215, 217, 227–228, 232–235
P0 and P1 registers
P0BR (P0 base register), 230–231
P0LR (P0 length register), 230–231
P1BR (P1 base register), 232–235
P1LR (P1 length register), 233–235
Packed decimal string. *See also* Decimal string
data type, 17–19
operations, 177
Page, defined, 214
Page boundary, access across, 226
Page-frame-number field (PTE(PFN)), 220, 422
Page protection, 214
memory management disabling and, 218
Page table entry (PTE), 220–222
cached, 235
changes to, 222–223
instruction stream modification and, 308, 310
invalid, 220
I/O device use of, 316–319
multiprocessing and, 223
system page table, 226
translation buffer for, 235–237
Page tables, 213, 217
PC (program counter), 19, 287
PCB (process control block), 287
PCBB (process control block base), 287–289
PDP-11. *See also* Compatibility mode
traps, unimplemented, 377
virtual address of, ix
Performance-monitor-enable (PME) register, 289
Perform self-test console command (TEST), 406
Per-process registers, context switching and, 323–327
Per-process space, 215
protection code and, 224
virtual address format and, 216
Per-process stack pointers, 287, 324
PFN (page-frame number field, PTE(PFN)), 220, 422
Physical address
generating from a P0 region virtual address, 232
generating from a P1 region virtual address, 234
generating from system region virtual address, 227
implementation options, 532–533
of process page tables, 229–230
SCB and, 269
translating from virtual addresses, 219
Physical memory, memory management disabling and, 217–219
PME (performance-monitor-enable) register, 289
POLY (polynomial evaluation instruction), floating-point, 130, 145–150
Polynomial evaluation instruction (POLY), floating-point, 145–150
POPR (pop registers instruction), 107
Powerfail, 384–385
Power-up initialization, after powerfail, 384–385
Precedence
of memory-management faults, 238, 499
of multiple events, 267
of trace faults, 262
Prefetching, instructions, 308
Previous access mode (PSL⟨PRV_MOD⟩), 23
Primary operand, in index mode addressing, 38
Priority level. *See* IPL
Privileged-instruction fault, defined, 259
Privileged internal processor registers (IPR). *See* IPR
Privileged registers. *See* IPR
Privileged services, 239–243
PROBE (probe instructions), 240–243
Procedure-call instructions, 91–98
CALLG (Call procedure with general argument list), 93–94
CALLS (call procedure with stack argument list), 95–96
RET (return from procedure), 97–98
Process
address-space, 215
address translation, 227–235
context, 272–273, 287–289
Process control block base (PCBB), process context, 287–289
Process control block (PCB), 287
process context, 287–289
Processor access modes. *See* Access modes
Processor
errors, 248
initialization, after powerfail, 384–385
page table entry and, 220–222
states, 19–24, 379
status, when exception or interrupt is serviced, 249
Processor status longword (PSL), 19, 21–23, 287
access mode and, 224
fields of, 22
Processor status word (PSW), 21
Process page table
addressing, 228–230
physical vs. virtual addressing of, 229–230
page table entry, 230
Process registers, compatibility mode, 377
Process-scheduling interrupts, 292
Process structure, 287–298
instructions, 292–298

Program counter (PC), 287
defined, 19
Program I/O mode, 413
Program (P0) region, 215, 217
Program synchronization, compatibility mode, 377–378
PROT, (protection field, PTE(PROT)), 221, 222, 224
Protection, checking, 240–241
Protection code, 224
PRV_MOD (previous access mode, PSL(PRV_MOD)), 23
PSL (processor status longword), 287
PSW (processor status word), 21
PTE (page table entry). *See* Page table entry
PUSH (push instructions)
PUSHA (push address), 68
PUSHL (push long), 63
PUSHR (push registers), 108

Quadruple precision floating point. *See* H_floating
Quadword
data type, 6
notation for, 26
in registers, 21
Queue
absolute, 11–12, 110–111
defined, 11, 110
empty, 11, 13
header of, 11, 13, 110–111
ill-formed, 110
operations on, 110–111
self-relative, 13, 110–111
tail of, 110
Queue instructions, 110–127
INSQHI (insert entry into queue at head, interlocked), 112–114
INSQTI (insert entry into queue at tail, interlocked), 115–117
INSQUE (insert entry in queue), 110–111, 118–119
interlocked, 111
REMQHI (remove entry from queue at head, interlocked), 120–122
REMQTI (remove entry from queue at tail, interlocked), 123–125
REMQUE (remove entry from queue), 126–127
Queue linkage longwords, 11, 13

READ access, 219
ordering of, 299–300
type, 26, 40. *See also* Access type
Read-only access, 224. *See also* Access type
Read-write access, 224. *See also* Access type
Region field, in virtual address format, 216
Register conflict, overlapped vector instruction execution and, 492
Register mode addressing, 28, 337
deferred indexed mode, 39
deferred mode, 29, 337–338
deferred mode operand specifier format, 29
operand specifier format, 28
Registers. *See also* Internal processor register (IPR); other specific registers
adapter configuration, 532
base, 19–20, 287
in character-string instructions, 153
compatibility mode, 337–340, 373
console terminal, 414–415
CPU identification (CPUID), 327
data types, 10, 20, 21
for decimal-string instructions, 175
general, compatibility mode, 337–340
general-purpose (GPR), 19–21
index, 19, 39
interval clock, 333–335
interval clock control status (ICCS), 333–335
interval count (ICR), 333–335
I/O, 319–321
next interval count (NICR), 333–334
per-process, 323–327
privileged, 323–335
system identification (SID), 330
system-type (SYS_TYPE), 330
time-of-year clock, 330–333
vector, 429–437, 492
vector control, 431–432
vector count (VCR), 431–432
vector length (VLR), 431
vector mask (VMR), 431
Register specifier fields, vector, 441
REI (return from exception or interrupt instruction), 249, 283–284
instruction stream modification and, 309–310
Relational operators, operation description notation and, 47
REM (remainder notation), 46
REM (remove queue instructions)
REMQHI (remove entry from queue at head, interlocked), 120–122, 304
REMQTI (remove entry from queue at tail, interlocked), 123–125, 304
REMQUE (remove entry from queue), 110–111, 126–127
Repeat console command (REPEAT), 405
REPEAT console command (repeat), 405
Replace sign when zero operator (EOSREPLACE_SIGN), EDITPC, 211
Reserved addressing mode fault, 39, 258
Reserved instruction fault, 128
compatibility mode, 375
defined, 259
Reserved operand exception (fault and abort)
ACB and, 75
BB and, 80, 81, 82
decimal-string instructions, 176–177
defined, 258–259
in BICPSW, 99
in BISPSW, 100
Reserved operands, floating-point numbers and, 129, 130, 461–463
Reserved to customer fault, 259–260
Restart, of system, 382–383
Restartability, after fault or interrupt, 249–250
Restart parameter block (RPB), 382–383
Restart-in-progress flag, 380, 382
RET (return from procedure instruction). *See* Return instructions
Return instructions
REI (return from exception or interrupt), 249, 283–284. *See also* REI
RET (return from procedure), 91, 92, 97–98
RSB (return from subroutine), 89
RTS (return from subroutine), 368
RTI (return from interrupt), 369, 373
RTT (return from trap), 369, 373

ROL (rotate left instruction), 353
ROR (rotate right instruction), 354
ROTL (rotate long instruction), 63
Rounding bit, 131–132
RPB (restart parameter block), 382–383
RSB (return from subroutine instruction), 89
RTI (return from interrupt instruction), 369
RTS (return from subroutine instruction), 368
RTT (return from trap instruction), 369
rtVAX
  memory management, 422–424
  variant, 229
Rubout console command, 389
RXCS (console receive control and status register), 414–415
RXDB (console receive data buffer register), 414–415

S (software field, PTE(S)), 221, 222
Save-process-context instruction (SVPCTX), 292, 296–298
SBC (subtract carry instruction), 351
SBR (system-base register), 226
SBWC (subtract with carry instruction), 64
SBZ (should be zero), 3
Scalar context switching, 487–488
Scalar floating-point instructions, reserved operands and, 130
Scalar processor
  exceptions generated by, 244
  synchronization with vector processor, 300, 509–520
  translation buffers in, 235, 236–237
  vector execution model and, 485–487
  vector processor disabled and, 501–503
Scalar/vector instruction synchronization (SYNC), 478, 509–510
Scan characters instruction (SCANC), character string, 168, 314
SCANC (scan characters instruction), character string, 168, 314
Scatter vector register data into memory instruction (VSCAT), 452–453
SCB (system control block), 251, 268–272
SCBB (system control block base), 268–270
Self-modifying code. *See* Instruction stream, modification
Self-relative queues, 110, 111
  data type, 13
Self-test (TEST) console command, 406
Serious system failure, 265–266
Service context, 273
SET console command, 405
SEXT (sign extend notation), 27, 46
Shared data
  atomicity and corruption of, 302–303
  interprocessor signaling and, 301
  preventing corruption of, with interlocks, 303–306
  VAX architecture and, 299
  visibility of, 301–302
Shift instructions. *See* Arithmetic shift instructions
SHOW console command, 405
Side effects, compatibility mode, 337
SID (system identification register), 330
Signaling, interprocessor, 299, 301–302
Sign bit, in floating-point notation, 128
Sign extend notation (SEXT), 27, 46
Sign extend word instruction (SXT), 352
Significance operator (EOS_SIGNIF), EDITPC, 212
Single operand instructions, compatibility mode, 345–354
Single-precision floating point. *See* F_floating
SIRR (software-interrupt-request-register), 252–253
SISR (software-interrupt-summary register), 252
SKPC (skip character instruction), character string, 169, 314
SOBx (subtract one and branch). *See* Subtract one and branch instructions
Software field (PTE(S)), 221, 222
Software-interrupt-request-register (SIRR), 252–253
Software interrupts, 251–253
Software-interrupt-summary register (SISR), 252
Source strings, memory references and, 313
SP. *See* Stack pointer (SP)
Span characters instruction (SPANC), character string, 170, 314
SPANC (span characters instruction), character string, 170, 314
Special characters, in console I/O mode, 388–390
SSP (supervisor-mode stack pointer), 273
SPT (system page table), 226–227
Stack, 272–276,
  alignment, 274
  not valid, 273–274
  overflow, 265
  overflow trap, compatibility mode, 377
  residency, 273–274
  running on interrupt stack, 244, 273–275, 292
  status bits, 274–275
  switching between, 273, 275
  top-of-stack, memory above, 276
Stack pointer (SP), 273
  accessing, 275–276, 327
  compatibility mode, 340
  definition of, 19
  images, per-process, 327
  in PCB, 275, 287–288
  IPR (internal processor register), 275, 324, 327
  per-process, 324, 327
Stack frame, 19
  call instructions, 92
  CHM instructions, 261
  emulation trap, 427
  machine-check exception, 266
  memory management fault, 238, 499
START console command (start execution), 406
Start execution console command (START), 406
State transitions, system, 385–387
Stop byte, 313
Store fill operator (EO$FILL), EDITPC, 206
Store sign operator (EO$STORE_SIGN), EDITPC, 212
Store vector register data into memory instruction (VST), 450–451
String descriptor, in CRC instruction, 171
String
  instruction interpretation, 41–42
  instructions, emulation of, 425
  leading-separate numeric, data type, 15–17
  numeric string instructions, 15, 185–191
  trailing numeric, data type, 13–15
SUB (subtract instruction). *See* Subtract instructions
Subroutine instructions, compatibility mode, 367–368
Subscript-range trap, defined, 256

Subsetting, 416–428
hardware goal, 416
implementation options, 421–424
instruction emulation, 424–428
instruction-set options, 416–421
software goal, 416
Subtract instructions
SBC (subtract carry), 351
SBWC (subtract with carry), 64
SUB (subtract), integer, 65
SUB (subtract), compatibility mode, 357
SUB (subtract), floating-point, 130, 151
SUBP (subtract packed), 196–197
Subtract one and branch instructions
SOB (subtract one and branch), 366
SOBGEQ (subtract one and branch greater than or equal), 89
SOBGTR (subtract one and branch greater than), 90
Supervisor access mode, 213, 224
Supervisor-mode stack pointer (SSP), 273
accessing, 327
Suspended-emulation fault, 426–428
SVPCTX (save-process-context instruction), 292, 296–298, 325
SWAB (swap bytes instruction), 354
Swap bytes instruction (SWAB), 354
SXT (sign extend word instruction), 352
SYNC (scalar/vector instruction synchronization), 478, 509–510
Synchronization, 509–520
compatibility mode, 377–378
memory activity completion synchronization (VMAC), 513–514
memory instruction (MSYNC), 511–513, 514, 515, 516
memory synchronization within the vector processor (VSYNC), 481,514–520
required use of memory synchronization instructions, 515–520
scalar/vector instruction (SYNC), 478, 509–510
scalar/vector memory, 510–512
Synchronize vector memory access instruction (VSYNC), 481
Synchronous memory management handling, 499–500
System-base register (SBR), 226
System bootstrapping. *See* Bootstrapping
System bus initialization, after powerfail, 384–385
System control block base (SCBB), 268–270
System control block (SCB), 251, 268–272
implementation options, 533
System failures, serious, 265–266
System-halted state (console I/O mode), 387–413
System identification register (SID), 330
System-length register (SLR), 227
System page table (SPT), 226–227
System-page-table length register (SLR), 230
System restart, 382–383
System running (program I/O mode), 413
System space, 215–216, 217
addresses, 226
address translation, 226–227
protection code and, 224
System state transitions, 385–387
SYS_TYPE (system-type register), 330

T bit (trace enable, PSL(T)), 24, 262
TB (translation buffer). *See* Translation buffer
TBx (translation-buffer IPRs). *See* Translation-buffer IPRs
TEST console command, 406
Test instruction (TST), 66, 348
floating-point, 130, 152
Time-of-year clock register, 330–333
Top-of-stack, memory above, 276
TP bit (trace pending, PSL(TP)), 22, 262
Trace enable bit (PSL(T)), 24, 262
Trace faults, 261, 262–265
compatibility mode, 375–377
Trace handlers, 263–265
Trace pending bit (PSL(TP)) 22, 262
Tracing
breakpoint fault, 261
in compatibility mode, 375–377
trace fault, 261, 262–265
Trailing numeric string, data type, 13–15
Transitions, between major system states, 385–387
Translation buffer (TB), 235–237
consistency, 235
flushing, 235
See also Translation-buffer IPRs
Translation-buffer IPRs (internal processor registers)
TBCHK (translation-buffer check), 237
TBIA (translation-buffer-invalidate-all), 235, 237
TBIS (translation-buffer-invalidate-single), 235, 237
VTBIA (vector translation buffer invalidate all), 432, 434
Translation-not-valid fault, 220, 226, 238, 280
defined, 257
stacks and, 273
TRAP instruction fault, compatibility mode, 375
Traps, defined, 246
TST (test instruction), 66, 348
floating-point, 130, 152
TXCS (console transmit control and status register), 414–415
TXDB (console transmit data buffer register), 414–415
Type
of processor, 330–332
of system, 330–332

ULTRIX, x
UNDEFINED operation, 2–3, 534–536
memory management disabling and, 219
Underflow
floating point, 23, 55, 131, 257
UNIBUS
architecture subsetting and, 424
address space, 532–533
device interrupts, 251
interrupt priority levels, 245
interrupt vectors, 272
I/O register restrictions and, 319–320
I/O structure and, 316
Unimplemented PDP-11 traps, 377
UNJAM console command, 384, 406
UNPREDICTABLE result, 2–3, 21, 533–534
aborts and, 247
in autodecrement mode addressing, 34
decimal-string instructions, 177
in immediate mode addressing, 29

in index mode addressing, 39
instruction interpretation and, 42
in page table entry, 221–222
in register mode addressing, 28
Unsigned numeric encoding, 15
Urgent interrupts, 251
User access mode, 213, 224
User-mode stack pointer (USP), 273
accessing, 327
USP (user-mode stack pointer). *See* user-mode stack pointer

V (overflow condition code, PSL(V)), 24
V (page-table entry valid bit, PTE(V)), 220–223
VADD (vector floating add instruction), 463
VADDL (vector integer add instruction), 454
Valid bit (PTE(V)), page-table entry, 220–223
Validation of arguments, 239–243
VAER (vector arithmetic exception register), 432–433, 436, 489
Variable-length bit field
base address access type, 26
data type, 9–11
in registers, 10, 21
Variable-length bit-field instructions, 69–73
CMP (compare field), 70
EXT (extract field), 71
FF (find first), 73
INSV (insert field), 69, 73
VAX architecture
32-bit, ix, xiii-xiv
subsetting, 416–428
vector architecture, 429
VAX implementations
MicroVAX 2000, 541
MicroVAX 3100, 543
MicroVAX 3300/3400, 542
MicroVAX 3500/3600, 542
MicroVAX 3800/3900, 542
MicroVAX-I, 538
MicroVAX-II, 539
VAX-11/725, 538
VAX-11/730, 537–538
VAX-11/750, 537
VAX-11/780, ix, 537
VAX-11/782, 538
VAX-11/785, 539
VAX 4000-300, 543
VAX 6000 Model, xiii, 400, 543
VAX 6200, 542
VAX 6300, 542
VAX 8200, 539
VAX 8250, 541
VAX 8300, 540
VAX 8350, 541
VAX 8500, 541
VAX 8530, 541
VAX 8550, 541
VAX 8600, 539
VAX 8650, ix, 540
VAX 8700, 541
VAX 8800, 540
VAX 8810/8820/8830/8840, 541
VAX 9000, xiii, xiv , 543
VAXstation 2000, 544
VAXstation 3100, 545, 546
VAXstation 3200/3500, 544–545
VAXstation 3520/3540, 545, 546
VAXstation 8000, 545, 546
VAXstation I, 544
VAXstation II, 544
VAXstation II/GPX, 544
VBIC, 459
VBIS, 459
VCMPL (vector integer compare instruction), 455–456
VCMP (vector floating compare instruction), 464–465
VCR (vector count registers), 431–432
VDIV (vector floating divide instruction), 468–469
Vector arithmetic exception register (VAER), 432–433, 436
Vector arithmetic exceptions, 500–501
Vector chaining, 489, 490–492
Vector context switching, 503–506
Vector control registers, 431–432, 493
Vector control word operand, 438
formats, 441–443
Vector convert instruction (VVCVT), 466–467
Vector count register (VCR), 431–432
Vector edit instructions, 474–476
IOTA (generate compressed iota vector), 475–476
VMERGE (vector merge), 474
Vector execution model, 485–498
Vector floating-point instructions, 461–473
reserved operands and, 130
VADD (vector floating add), 463
VCMP (vector floating compare), 464–465
VDIV (vector floating divide), 468–469
VMUL (vector floating multiply), 470–471
VSUB (vector floating subtract), 472–473
VVCVT (vector convert), 466–467
Vector hardware errors, 521–522
Vector instruction exceptions, 23
Vector instruction formats, 438–444
exception enable bit, 439–440
masked operations, 438–439
modify intent bit, 440
register specifier fields, 441
restrictions on operand specifier usage, 443–444
vector control word formats, 441–443
Vector instructions, 23, 420–421
assembler notation, 482–484
edit, 474–476
floating-point, 461–473
integer, 454–458
logical, 459–460
memory access, 444–453
miscellaneous, 477–481
overlapped execution, 489–498
shift, 459–460
Vector instruction set, classes, 444
Vector integer instructions, 454–458
VADDL (vector integer add), 454
VCMPL (vector integer compare), 455–456
VMULL (vector integer multiply), 457
VSUBL (vector integer subtract), 458
Vector length register (VLR), 431
Vector logical functions instruction (VxlogL), 459–460
Vector mask register (VMR), 431, 438–439, 492
Vector memory access instructions, 444–453
VGATH (gather memory data into vector register), 448–449

Vector memory access instructions *(continued)*
VLD (load memory data into vector register), 446–447
VSCAT (scatter vector register data into memory), 452–453
VST (store vector register data into memory), 450–451
Vector memory activity check (VMAC), 432, 434, 437
Vector memory management, 520–521
disabled, 521
Vector merge instruction (VMERGE), 474
Vector opcodes, illegal, 444
Vector operate instructions, 430
Vector processor, 23, 420–421
access mode restrictions, 487
dependencies among vector results and, 493–498
disabled, 501–506
exceptions, 244, 498–509
memory management exception handling, 498–500
register conflict and, 492
synchronization with scalar processor, 300, 509–520
translation buffers in, 235, 236–237
vector execution mode and, 485–487
Vector processor status register (VPSR), 432–433, 436–437
Vector registers, 430–437
dependencies among vector results and, 493–498
register conflict, 492
support for console access of, 410
Vectors in the SCB
exception, 270–272
interrupt, 270–272
Vector shift logical instruction (VSL), 460
Vector state address register (VSAR), 432, 435–436
Vector translation buffer invalidate all (VTBIA) IPR, 235, 432, 434
VGATH (gather memory data into vector register instruction), 448–449, 500
Virtual address, 4
defined, 214
format, 216
length violation in, 224–225
per-process space, 215
process page tables, 229–230
SCB and, 269
space, ix, 214–217
system, 215–216, 226–227
translation, 219, 226–227
Virtual memory system, 213, 214
Virtual page number (VPN) field, 216
Visibility of data, 301–302
VLD (load memory data into vector register instruction), 446–447, 500
VLR (vector length register), 431
VMAC (vector memory activity check), 432, 434, 437, 513–514
hardware errors and, 521–522
VMB, 380–381
VMERGE (vector merge instruction), 474
VMR (vector mask register), 431, 438–439, 492
VMS, x
VMUL (vector floating multiply instruction), 470–471
VMULL (vector integer multiply instruction), 457
VPN (virtual page number) field, 216
VPSR (vector processor status register), 432–433, 436–437, 488, 500
VSAR (vector state address register), 432, 435–436
VSCAT (scatter vector register data into memory instruction), 452–453, 500
VSL (vector shift logical instruction), 460
VSMERGEx pseudo-opcodes, 484
VSPR
hardware errors and, 521–522
vector processor disabled faults and, 503–506
VSTL, 500
VST (store vector register data into memory instruction), 450–451
VSUB (vector floating subtract instruction), 472–473
VSUBL (vector integer subtract instruction), 458
VSYNC (synchronize vector memory access instruction), 481, 514–515
when not required, 517–520
VTBIA (vector translation buffer invalidate all) IPR, 235, 432, 434
VVCVT (vector convert instruction), 466–467
VxlogL (vector logical functions instruction), 459–460
VXOR, (vector exclusive-or), 459–460

Word branch mode addressing, 39–40
Word data type
defined, 4–5
notation for, 26
in registers, 20
Word displacement mode addressing, 34–35
deferred indexed mode, 39
deferred mode operand specifier format, 35
indexed mode, 39
operand specifier format, 34–35
Word tearing, 302
Write access, 219
ordering of, 299–300
Write access type, 26, 40. *See also* Access type

X console command, 407–408
XFC (extended function call instruction), 109
fault, 259–260
XOR (exclusive-OR instruction), 66
compatibility mode, 360
See also VXOR

Z (zero condition code, PSL⟨Z⟩), 24
Zero-length decimal strings, 177–178
Zero numbers, in decimal-string instructions, 176
ZEXT (zero extend notation), 27, 46
Zoned numeric encoding, 15

VAX Architecture Reference Manual
Second Edition

Compiled by Digital's own VAX engineers, this definitive reference manual describes the architecture of a VAX processor as seen by a machine language programmer and as implemented consistently across the entire family of VAX systems—from the original VAX 11/780 to the VAXstation 3100 and the VAX 9000.

This new Second Edition describes recently added features, such as VAX vector processing, expanded physical memory support, and the refined shared memory model supported in VAX multiprocessor systems. Chapters cover the data types, instructions, addressing modes, registers, exceptions, interrupts, memory management, process structure, and programming practices common to all VAX systems.

Richard A. Brunner is a Principal Engineer and current VAX Architect for Digital Equipment Corporation. He holds an M.S.E.E. in Computer Engineering from Rensselaer Polytechnic Institute.

Also from Digital Press

VMS File System Internals. Kirby McCoy
1990. Softbound. Order number EY-F575E-DP

VAX/VMS Internals and Data Structures: Version 5.2
Ruth E. Goldenberg and Lawrence J. Kenah. 1991. Hardbound.
Available Spring 1991.

digital™
Digital Press
12 Crosby Drive
Bedford, Massachusetts 01730

ORDER NUMBER EY-F57
ISBN 1-55558-
PH ISBN 0-13-92